清华大学研究生公共课教材——数学系列

《最优化基础——模型与方法》系列教材

网络优化（第2版）

谢金星 邢文训 王振波 编著

清华大学出版社

本书系统介绍了网络优化的基本模型和基本算法。在模型方面，主要介绍树的问题（包括最小树、最小树形图、最大分枝）、最短路问题、最大流问题、最小费用流问题和匹配问题等。在算法方面，针对不同的模型选择了许多各具特色的算法，其中既有比较经典的算法，也有比较新颖、实用的算法。

ISBN 978-7-302-20325-4

9 787302 203254 >

定价：19.00元

清华大学研究生公共课教材——数学系列

《最优化基础——模型与方法》系列教材

网络优化(第2版)

谢金星 邢文训 王振波 编著

清华大学出版社
北京

内容简介

本书系统介绍了网络优化的基本模型和基本算法，包括构造这些算法的基本思想以及相应算法在计算机上的一些具体实现技巧和复杂性分析.

全书由7章组成：第1章为概论，第2章介绍关于算法的一些基本知识，第3章到第7章分别讨论树的问题、最短路问题、最大流问题、最小费用流问题和匹配问题. 每章还安排了一些练习题.

本书可作为数学、应用数学、运筹学、管理科学、系统科学、信息科学、计算机科学与工程等专业的高年级大学生和研究生教材，也可供其他相关专业的学者和技术人员参考.

图书在版编目（CIP）数据

网络优化/谢金星，邢文训，王振波编著. —2版. —北京：清华大学出版社，2009.7
（清华大学研究生公共课教材. 数学系列）
ISBN 978-7-302-20325-4

Ⅰ. 网… Ⅱ. ①谢… ②邢… ③王… Ⅲ. 网络模型—最优化算法—研究生—教材
Ⅳ. O224

中国版本图书馆CIP数据核字(2009)第093535号

责任编辑：刘 颖
责任校对：王淑云
责任印制：杨 艳

出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　编**：100084
社 总 机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：北京季蜂印刷有限公司
装 订 者：三河市李旗庄少明装订厂
经　　销：全国新华书店
开　　本：185×230　**印　张**：11.25　**字　数**：245千字
版　　次：2009年7月第2版　**印　次**：2009年7月第1次印刷
印　　数：1～4000
定　　价：19.00元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换.
联系电话：010-62770177转3103　　产品编号：033496-01

序　言

最优化是人们在工程技术、科学研究和经济管理等诸多领域中经常遇到的问题. 结构设计要在满足强度要求等条件下使所用材料的总重量最轻；资源分配要使各用户利用有限资源产生的总效益最大；安排运输方案要在满足物质需求和装载条件下使运输费用最低；编制生产计划要按照产品工艺流程和顾客需求，尽量降低人力、设备、原材料等成本使总利润最高. 可以预测，随着科学技术尤其是计算机技术的不断发展，以及数学理论与方法向各学科和各应用领域更广泛、更深入的渗透，在 21 世纪这个信息时代，最优化理论和技术必将在社会的诸多方面起着越来越大的作用.

解决实际生活中优化问题的手段大致有以下几种：一是靠经验的积累，凭主观作出判断；二是做试验选方案，比优劣定决策；三是建立数学模型，求解最优策略. 虽然由于建模时要作适当的简化，可能使结果不一定非常完善，但是它基于客观数据，求解问题简便、经济，而且规模可以很大(将来会越来越大). 人们还可以吸收从经验得到的规则，用实验不断校正建立的模型. 随着数学方法和计算机技术的进步，用建模和数值模拟解决优化问题这一手段，将会越来越显示出它的效能和威力. 显然，在决策定量化、科学化的呼声日益高涨的今天，数学建模方法的推广应用是符合时代潮流和形势发展需要的.

最优化理论、模型与方法所包含的内容很多，国内已出版了不少教材和专著介绍其各个分支. 但是，一方面，近年来发展起来的、有着广泛应用背景的规划模型(如随机规划、模糊规划等)，以及一些已经为许多人采用、受到广泛关注的优化算法(如模拟退火算法、遗传算法等)还缺乏详细和系统的介绍；另一方面，一些偏重优化理论和方法的教材，其要求难以与工科学生的数学知识衔接，也缺少对于应用来说十分重要的建模过程和软件介绍，而一些比较通俗的运筹学教材，则在加强理论基础，适应学生将来从事科研工作需要上考虑较少. 这套教材试图弥补以上两方面的缺陷，力求体现下述特点：

1. 内容既包含传统的线性规划与非线性规划等部分，又纳入有广泛应用前景的随机规划和模糊规划；在传统内容中，既注重典型的数学思想和方法的系统叙述，又引入丰富的建模实例.

2. 数学基础既与工科学生所学知识衔接，又考虑到研究生阅读文献、从事科研工作的需要，适当提高理论基础的起点.

3. 对一般教材介绍的诸多算法进行精选，配合介绍一些应用软件，并引入近年来迅速发展的若干新算法.

本系列教材将陆续出版，首批四册为《线性与非线性规划》、《网络优化》、《现代优化计算方法》、《随机规划与模糊规划》.

由于水平有限，书中难免有缺陷和错误，诚恳希望读者予以批评指正.

《最优化基础——模型与方法》系列教材编委会

2005 年 3 月

前　言

我们生活在一个网络社会中. 从某种意义上说,现代社会是一个由计算机信息网络、电话通信网络、运输服务网络、能源和物质分派网络等各种网络所组成的复杂的网络系统. 网络优化就是研究如何有效地计划、管理和控制这个网络系统,使之发挥最大的社会和经济效益.

网络优化是运筹学(Operations Research)中的一个经典和重要的分支,所研究的问题涉及物资管理、经济管理、工业工程、交通运输、计算机科学与信息技术、通讯与网络技术、控制论及军事运筹学等诸多领域. 因此学习一些有关网络优化的基本知识,对许多专业的学生和许多领域的科技人员、管理人员都是相当必要的.

本书系统介绍了网络优化的基本模型和基本算法. 在基本模型方面,主要介绍树的问题(包括最小树、最小树形图、最大分枝)、最短路问题、最大流问题、最小费用流问题和匹配问题等. 在基本算法方面,由于处理上述这些问题的算法非常多,而且新算法还在不断涌现,从大量算法中选择哪些典型算法进行介绍本身是见仁见智的事,我们只能有选择地介绍其中部分算法. 我们既介绍一些比较经典的算法,也介绍一些比较新颖、实用的算法.

本书由 7 章组成. 第 1 章为概论,主要介绍图和网络的一些基本概念、图和网络在计算机上的表示方法,并初步介绍计算复杂性理论. 第 2 章介绍关于算法的一些基本知识,包括计算复杂性理论、近似算法、整数规划、动态规划,以及一些常用的网络搜索算法等. 第 3 章到第 7 章分别介绍树的问题、最短路问题、最大流问题、最小费用流问题和匹配问题,这是网络优化的基本内容.

本次成书过程中参考了大量国内外有关文献和教材,并对内容进行了进一步的筛选和扩充. 我们希望本书能适合于各个层次的读者阅读. 对于数学和计算机等理论性要求较高的专业的教学,最好能讲授本书全部或大部分内容,并要求读者在学习本课程之前已经掌握了线性规划的基础知识. 我们希望不仅能让这部分读者了解各种网络优化算法的基本思想和基本理论,而且能了解一些实现技巧并学会算法的复杂性分析方法. 在各章的练习题中,我们还配备了一定数量难度较高的题目作为对书中讲授内容的扩充. 对于其他理论性要求不太高的专业的教学或不太关心算法复杂性的读者来说,可以在阅读中略去部分理论内容. 对于只想对网络优化有所了解或只希望从本书中查询一些具体算法的读者来说,完全可以略去全部的理论部分,而只了解相应的模型和算法就可以了.

自 2000 年以来,笔者以本书为教材在清华大学讲授研究生课程"网络优化". 在此期间,得到多方面的反馈和宝贵意见. 在此基础上,我们对本书的结构和内容进行了一些调整和修

改.在第2版中增加了算法基础一章,介绍一些准备知识,包括复杂性理论、近似算法及一些常用的算法.第1版中的整数规划和动态规划两章不再作为单独的章节出现,一些相关的内容安排在算法基础一章.

最后,对我们家人的理解、耐心和支持表示由衷的谢意!由于我们的水平有限,恳请读者对本书的不足之处提出批评指正.

谢金星　邢文训　王振波

2009年春于清华园

目　录

第1章 概　论

我们生活在一个网络社会中.从某种意义上说,现代社会是一个由计算机信息网络、电话通信网络、运输服务网络、能源和物质分派网络等各种网络所组成的复杂的网络系统.网络优化就是研究如何有效地计划、管理和控制这个网络系统,使之发挥最大的社会和经济效益.

网络优化是运筹学(Operations Research)中的一个经典和重要的分支,所研究的问题涉及经济管理、工业工程、交通运输、计算机科学与信息技术、通讯与网络技术等诸多领域.本书中将要讨论的最短路问题、最大流问题、最小费用流问题和匹配问题等都是网络优化的基本问题.

本章主要介绍网络优化问题的一些实际例子以及图与网络的基本概念,初步介绍计算复杂性理论,为后续章节的学习奠定基础.

1.1 网络优化问题的例子

我们首先通过一些例子来了解网络优化问题.

例 1.1 公路连接问题

某地区有若干个主要城市,现准备修建高速公路把这些城市连接起来,使得从其中任何一个城市都可以经高速公路直接或间接到达另一个城市.假定已经知道了任意两个城市之间修建高速公路的成本,那么应如何决定在哪些城市间修建高速公路,使得总成本最小? □

例 1.2 最短路问题(shortest path problem,SPP)

一名货柜车司机奉命在最短的时间内将一车货物从甲地运往乙地.从甲地到乙地的公路网纵横交错,因此有多种行车路线,这名司机应选择哪条线路呢?假设货柜车的运行速度是恒定的,那么这一问题相当于需要找到一条从甲地到乙地的最短路. □

例 1.3 运输问题(transportation problem)

某种原材料有 M 个产地,现在需要将原材料从产地运往 N 个使用这些原材料的工厂假定 M 个产地的产量和 N 家工厂的需要量已知,单位产品从任一产地到任一工厂的运费已知,那么如何安排运输方案可以使总运输成本最低? □

例 1.4 指派问题(assignment problem)

一家公司经理准备安排 N 名员工去完成 N 项任务,每人一项.由于各员工的特点不同,不同的员工去完成同一项任务时所获得的回报是不同的.如何分配工作方案可以使总回

报最大？ □

例 1.5 中国邮递员问题(chinese postman problem,CPP)

一名邮递员负责投递某个街区的邮件.如何为他(她)设计一条最短的投递路线(从邮局出发,经过投递区内每条街道至少一次,最后返回邮局)？由于这一问题是我国管梅谷教授1960年首先提出的,所以国际上称之为中国邮递员问题. □

例 1.6 旅行商问题(traveling salesman problem,TSP)

一名推销员准备前往若干城市推销产品.如何为他(她)设计一条最短的旅行路线(从驻地出发,经过每个城市恰好一次,最后返回驻地)？这一问题的研究历史十分悠久,通常称之为旅行商问题. □

上述问题有两个共同的特点:一是它们的目的都是从若干可能的安排或方案中寻求某种意义下的最优安排或方案,数学上把这种问题称为最优化或优化(optimization)问题;二是它们都易于用图形的形式直观地描述和表达,数学上把这种与图相关的结构称为网络(network).与图和网络相关的最优化问题就是网络最优化或称网络优化(network optimization)问题.所以上面例子中介绍的问题都是网络优化问题.由于多数网络优化问题是以网络上的流(flow)为研究的对象,因此网络优化这门课程在许多学校里又常常被称为网络流(network flows)或网络流规划等.

下面首先简要介绍图与网络的一些基本概念.

1.2 图与网络

1.2.1 有向图与网络的基本概念

定义 1.1 一个有向图(directed graph 或 digraph) G 是由一个非空有限集合 $V(G)$ 和 $V(G)$ 中某些元素的有序对集合 $A(G)$ 构成的二元组,记为 $G=(V(G),A(G))$. 其中 $V(G)=\{v_1,v_2,\cdots,v_n\}$ 称为图 G 的节点集(node set)或顶点集(vertex set),$V(G)$ 中的每一个元素 v_i ($i=1,2,\cdots,n$) 称为该图的一个节点(node)或顶点(vertex); $A(G)=\{a_1,a_2,\cdots,a_m\}$ 称为图 G 的弧集(arc set),$A(G)$ 中的每一个元素 a_k(即 $V(G)$ 中某两个元素 v_i,v_j 的有序对)记为 $a_k=(v_i,v_j)$ 或 $a_k=v_iv_j$ ($k=1,2,\cdots,m$),被称为该图的一条从 v_i 到 v_j 的弧(arc).在不引起混淆的情况下,记号 $V(G)$ 和 $A(G)$ 中也可以省略 G,即分别记节点集、弧集为 V 和 A,而记有向图 $G=(V,A)$.

如果对有向图 G 中的每条弧赋予一个或多个实数,得到的有向图称为赋权有向图或有向网络,简称为网络(network).为了讨论方便,本书对图和网络不作严格区分,因为任何图总是可以赋权的.

当弧 $a_k=(v_i,v_j)$ 时,称 v_i,v_j 为弧 a_k 的端点(endpoint),其中 v_i 为尾(tail),v_j 为头(head),并称 v_j 在 G 中与 v_i 相邻(adjacent)或 v_j 是 v_i 的邻居(neighbor);弧 a_k 称为与节点

v_i, v_j 关联(incident),并称弧 a_k 为 v_i 的出弧(outgoing arc),为 v_j 的入弧(incoming arc).如果某两条弧至少有一个公共端点,则称这两条弧在图 G 中相邻.

图的概念是现实生活中图形概念的数学抽象,因此一个图也可以用图形来直观表示:用小圆圈表示节点,用节点间带箭头的连线表示弧及其方向(箭头从弧的尾指向弧的头).例如,图 1.1 的图形表示的是有向图 $G=(V,A)$ 其中节点集 $V=\{v_1,v_2,v_3,v_4,v_5\}$,弧集 $A=\{a_1,a_2,a_3,a_4,a_5,a_6\}$,弧 $a_1=(v_1,v_2)$,$a_2=(v_1,v_2)$,$a_3=(v_2,v_3)$,$a_4=(v_3,v_4)$,$a_5=(v_4,v_1)$,$a_6=(v_3,v_3)$.在该图中,弧 a_1,a_2 都是从节点 v_1 指向节点 v_2,这种弧称为多重弧(multiarc),相应的图称为多重图(multi-graph).值得注意的是:这时弧集合 A 也应当理解成多重集(multi-set),即 A 中可以包括多个相同的元素,这与我们通常所说的普通集合的概念有所不同.在该图中,弧 a_6 的头和尾都相同(v_3),这种弧称为环(loop).在该图中,没有任何弧与节点 v_5 关联,这样的节点称为孤立点(isolated vertex).

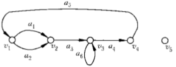


图 1.1 有向图的一般例子

图中节点的个数称为图的阶(order).通常用 $|V|$ 或 n 表示顶点个数(即图的阶数),用 $|A|$ 或 m 表示弧的条数.特别地,我们称只有一个节点的图为平凡图(trivial graph);不包括任何弧的图为空图(null graph).没有环、且没有多重弧的图称为简单图(simple graph).例如,在图 1.1 所示的图 G 中,图的阶数为 $n=|V|=5$,弧的条数为 $m=|A|=6$,图 G 不是平凡图或空图,也不是简单图.如不特别说明,本书中以后假设讨论的都是简单图.

图中与一个节点关联的出弧的数目称为该节点的出度(outdegree),入弧的数目称为该节点的入度(indegree),而出度与入度之和称为该节点的度(degree).度数为偶数的节点称为偶点(even point),否则称为奇点(odd point).例如,在图 1.1 所示的图 G 中,节点 v_2 的度为 3,一般记为 $d_G(v_2)=3$;节点 v_2 的出度为 1,一般记为 $d_G^+(v_2)=1$;节点 v_2 的入度为 2,一般记为 $d_G^-(v_2)=2$.在不引起混淆的情况下,也可以省略 G,即分别记为 $d(v_2)=3$;$d^+(v_2)=1$;$d^-(v_2)=2$.图 1.1 所示的图 G 中节点 v_1,v_2 为奇点,节点 v_3,v_4,v_5 为偶点.

假设 $G'=(V',A')$ 和 $G=(V,A)$ 是两个有向图,如果 $V'\subseteq V, A'\subseteq A$,则图 $G'=(V',A')$ 称为图 $G=(V,A)$ 的子图(subgraph),可简记为 $G'\subseteq G$.任给 $V'\subseteq V$,以 V' 为节点集的 G 的最大子图定义为:节点集为 V',弧集为两个端点都在 V' 中的 G 中所有弧的集合.以 V' 为节点集的 G 的最大子图称为 G 的由 V' 导出的子图(induced subgraph),即如果 $V'\subseteq V, A'=\{(v_i,v_j)\in A \mid v_i,v_j\in V'\}$,则图 $G'=(V',A')$ 称为图 $G=(V,A)$ 的由 V' 导出的子图(简称节点导出子图),记为 $G[V']$.类似地,可以定义弧子集 $A'\subseteq A$ 导出的子图(简称弧导出子图)$G[A']=(V',A')$,其中 $V'=\{v\in V \mid \exists v'\in V$,使得 $(v,v')\in A'$ 或 $(v',v)\in A'\}$.由此可以看出,给定 $V'\subseteq V$(或 $A'\subseteq A$),图 $G(V,A)$ 的导出子图是唯一的.例如,在图 1.1 所示的图 G 中,$G[\{v_1,v_2\}]=(\{v_1,v_2\},\{a_1,a_2\})$,$G[\{a_1,a_3\}]=(\{v_1,v_2,v_3\},\{a_1,a_3\})$,$G[\{a_1,a_3,a_5\}]=(\{v_1,v_2,v_3,v_4\},\{a_1,a_3,a_5\})$.

图 $G=(V,A)$ 的支撑子图（spanning subgraph，又称生成子图）是包含 G 的所有节点的子图，即当 $V'=V,A'\subseteq A$ 时，$G'=(V',A')$ 称为 $G=(V,A)$ 的支撑子图. 由此可以看出，图 $G=(V,A)$ 的支撑子图一般是不唯一的. 例如，在图 1.1 所示的图 G 中，$(\{v_1,v_2,v_3,v_4,v_5\},\{a_1,a_3,a_5\})$ 和 $(\{v_1,v_2,v_3,v_4,v_5\},\{a_1,a_2,a_5,a_6\})$ 等都是 G 的支撑子图.

有向图中的途径（walk）是该图的一些节点 $i_1,i_2,\cdots,i_r$ 和弧 $a_1,a_2,\cdots,a_{r-1}$ 所组成的子图，这些节点与弧可以交错排列成点弧序列 $i_1,a_1,i_2,a_2,\cdots,a_{r-1},i_r$，其中 $a_k=(i_k,i_{k+1})$ 或 $a_k=(i_{k+1},i_k)(\forall k=1,2,\cdots,r-1)$. 如果该序列中的所有弧都指向同一方向，即 $a_k=(i_k,i_{k+1})(\forall k=1,2,\cdots,r-1)$，则该途径称为有向途径（directed walk）. 在不引起混淆的情况下（如在简单图的情形下），可以仅仅用节点序列 $i_1,i_2,\cdots,i_r$ 或弧序列 $a_1,a_2,\cdots,a_{r-1}$ 来表示途径. 我们称 $r-1$ 为途径的长，分别称 i_1,i_r 为途径的起点和终点.

有向图中的路（path）是该图的不包含重复节点的途径. 路中的弧可以分成两类：其中一类弧 $a_k=(i_k,i_{k+1})$ 的方向与路的起点到终点的方向一致，称为路的前向弧（forward arc）；另一类弧 $a_k=(i_{k+1},i_k)$ 的方向与路的起点到终点的方向不一致，称为路的后向弧或反向弧（backward arc）. 路 P 的所有前向弧的集合一般记为 P^+，反向弧的集合记为 P^-. 有向图中的有向路（directed path）是该图的不包含重复节点的有向途径，即不包含反向弧的路. 例如，在图 1.1 所示的图 G 中，$v_1\ a_5\ v_4\ a_4\ v_3$ 就是从节点 v_1 到 v_3 的一条路，但不是从节点 v_1 到 v_3 的有向路，因为路上的弧都是反向弧；而 $v_3\ a_4\ v_4\ a_5\ v_1$ 就是从节点 v_3 到 v_1 的一条有向路. 又如，$v_2\ a_3\ v_3\ a_4\ v_4$ 是从节点 v_2 到 v_4 的一条有向路.

有向图 $G=(V,A)$ 中，如果 $(i_1,i_r)\in A$ 或 $(i_r,i_1)\in A$，则路 $i_1,i_2,\cdots,i_r$ 加上弧 (i_1,i_r) 或 (i_r,i_1) 组成的途径称为该有向图的圈（cycle），即圈是起点和终点重合且不含其他重复节点的途径. 同路的前向弧和反向弧的定义类似，可以定义圈的前向弧和反向弧. 有向图中的有向圈（directed cycle）是该图的不包含反向弧的圈，即由路 $i_1,i_2,\cdots,i_r$ 加上弧 (i_r,i_1) 组成. 如在图 1.1 所示的图 G 中，$v_1\ a_1\ v_2\ a_2\ v_1$ 就是一个圈，但不是有向圈，而 $v_1\ a_1\ v_2\ a_3\ v_3\ a_4\ v_4\ a_5\ v_1$ 就是一个有向圈. 不包含有向圈的图称为无圈图（acyclic graph）. 因此，图 1.1 所示的图 G 不是无圈图.

对于有向图 $G=(V,A)$ 中的两个节点，如果在图中至少存在一条路把它们连接起来，则称这两个节点是连通的（connected）. 如果图中任意两个节点都是连通的，则称该图为连通的；否则称为不连通的（disconnected）. 若 $G=(V,A)$ 不连通，$G'=(V',A')$ 是 $G=(V,A)$ 的连通子图，且不存在 $G=(V,A)$ 的连通子图 $G_1=(V_1,A_1)$ 满足 $G'\subseteq G_1,G'\neq G_1$，则称 $G'=(V',A')$ 为 G 的连通分支（component）. 也就是说，连通分支是图中的极大连通子图，不连通图可以分解成一些连通分支的并，而连通图只有一个连通分支. 例如，图 1.1 所示的图 G 是不连通的，G 包括两个连通分支，其中一个是只包括节点 v_5 的空图，另一个是除去节点 v_5 后所有节点和弧所构成的子图.

如果有向图中从任意一个节点出发，都存在至少一条有向路到达任意另一个节点，则称该图为强连通的（strongly connected）. 同样可以与连通分支类似地定义强连通分支. 例如，

图 1.1 所示的图 G 除去节点 v_5 后所有节点和弧所构成的子图是强连通的(因此,这样的子图不仅是 G 的连通分支,也同时是一个强连通分支).

设 S,T 是节点集合 V 的一个划分(即 $S,T\subseteq V;S,T\neq\varnothing;S\cup T=V;S\cap T=\varnothing$),则称两个端点分别位于 S,T 的弧为一个割(cut),记为$[S,T]=\{(i,j)\in A\mid i\in S,j\in T\}\cup\{(j,i)\in A\mid i\in S,j\in T\}$. 例如,在图 1.1 所示的图 G 中,若 $S=\{v_1,v_4,v_5\}$,$T=\{v_2,v_3\}$,则$[S,T]=\{a_1,a_2,a_4\}$.

1.2.2 无向图与无向网络的基本概念

有时我们关心的只是两个节点之间的联系,而不关心这种联系的方向性,这样就自然得到了无向图和无向网络的概念.

定义 1.2 一个无向图(undirected graph)G 是由一个非空有限集合 V 和 V 中某些元素的无序对集合 E 构成的,即 $G=(V,E)$. 其中 $V=V(G)=\{v_1,v_2,\cdots,v_n\}$仍称为图 G 的节点集(node set)或顶点集(vertex set),V 中的每一个元素 $v_i(i=1,2,\cdots,n)$ 称为该图的一个节点(node)或顶点(vertex);$E=E(G)=\{e_1,e_2,\cdots,e_m\}$通常称为图 G 的边集合(edge set),E 中的每一个元素 e_k(即 V 中某两个元素 v_i,v_j 的无序对) 记为$e_k=(v_i,v_j)$或 $e_k=v_iv_j$,或 $e_k=v_jv_i(k=1,2,\cdots,m)$,称为该图的一条边(edge). 边上赋权的无向图称为赋权无向图或无向网络(undirected network). 同有向图情况一样,本书对无向图和无向网络也不作严格区分.

有向图的其他相关概念都可以自然地推广到无向图上来. 自然,有时也会有一些微小的差异:如无向图的一条边的端点不再有头和尾之分;无向图的一个节点的度也不再有入度和出度之分;无向图中的一条路(或一个圈)上的一条边也不再有前向和反向之分;无向图的连通性不再有所谓的强连通概念;等等.

一个无向图也可以用图形来直观表示:用小圆圈表示节点,用节点间的连线表示边. 如图 1.2 的图形表示的是无向图 $G=(V,E)$,其中 $V=\{v_1,v_2,v_3,v_4,v_5\}$, $E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$,$e_1=(v_1,v_2)$,$e_2=(v_1,v_2)$,$e_3=(v_2,v_3)$,$e_4=(v_3,v_4)$,$e_5=(v_1,v_4)$,$e_6=(v_3,v_3)$. 这个图显然是图 1.1 所示的有向图中忽略弧的方向而得到的,有人把这样得到的无向图称为原有向图的基础图(underlying graph).

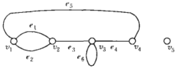


图 1.2 无向图的一般例子

下面再介绍两种特殊的图. 若无向图 G 的任意两个节点间有且只有一条边相连,则称其为完全图(complete graph). n 阶完全图记为 K_n. 若图 $G=(V,E)$的节点集 V 可以表示成两个互不相交的非空子集 S,T 的并集,且使得 G 中每一条边的一个端点在 S 中,另一端点在 T 中,则称 G 为二部图(bibartite graph). 当$|S|=p,|T|=q$,且 S 与 T 中任意两对节点都相邻时,称 G 为完全二部图,记为 $K_{p,q}$. 类似地,也可以定义有向的完全图和有向的完全二部图.

其他与图相关的概念还有很多，其中有些概念我们会在后续章节中用到时再作介绍. 对图论有兴趣的读者可以参考图论方面的专门书籍，这里就不详细介绍了. 值得提请读者注意的是，有关图的术语目前并不完全统一，不同的作者可能用不同的术语表达同一个概念，也可能使用的术语相同，表达的概念却并不完全相同. 因此在阅读有关图和网络的书籍和文献时，应注意防止由此而可能引起的误解. 此外，本书中在不引起混淆的情况下，有时将有向图和无向图都简称为图(graph)，具体所指应可从上下文中看出. 并且，我们也不再对弧和边的概念作严格区分，而认为它们表示相同的概念.

1.3 图与网络的数据结构

网络优化研究的是网络上的各种优化模型与算法. 为了在计算机上实现网络优化的算法，首先我们必须有一种方法(即数据结构)在计算机上描述图与网络. 一般来说，算法的好坏(即我们后面将要介绍的复杂度概念)与网络的具体表示方法，以及中间结果的操作方案是有关系的. 这里我们介绍计算机上用来描述图与网络的 5 种常用表示方法：邻接矩阵表示法、关联矩阵表示法、弧表表示法、邻接表表示法和星形表示法. 在下面的讨论中，我们首先假设 $G=(V,A)$ 是一个简单有向图，$|V|=n$，$|A|=m$，并假设 V 中的节点用自然数 1，2，…，n 表示或编号，A 中的弧用自然数 1，2，…，m 表示或编号. 对于有多重边或无向网络的情况，我们只是在讨论完简单有向图的表示方法之后，给出一些说明.

1.3.1 邻接矩阵表示法

邻接矩阵表示法是将图以邻接矩阵(adjacency matrix)的形式存储在计算机中. 图 $G=(V,A)$ 的邻接矩阵 $\boldsymbol{C}$ 是如下定义的：$\boldsymbol{C}$ 是一个 $n\times n$ 的 0-1 矩阵，即

$$\boldsymbol{C}=(c_{ij})_{n\times n}\in\{0,1\}^{n\times n},$$

$$c_{ij}=\begin{cases}0, & (i,j)\notin A,\\ 1, & (i,j)\in A.\end{cases}$$

也就是说，如果两节点之间有一条弧，则邻接矩阵中对应的元素为 1；否则为 0. 每行元素之和正好是对应节点的出度，每列元素之和正好是对应节点的入度. 可以看出，这种表示法非常简单、直接. 但是，在邻接矩阵的所有 n^2 个元素中，只有 m 个为非零元. 如果网络比较稀疏，这种表示法浪费大量的存储空间，从而增加了在网络中查找弧的时间.

例如，图 1.3 所示的网络图，可以用邻接矩阵表示为

$$\begin{bmatrix}0&1&1&0&0\\0&0&0&1&0\\0&1&0&0&0\\0&0&1&0&1\\0&0&1&1&0\end{bmatrix}.$$

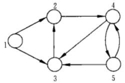


图 1.3 有向简单图的例子

同样，对于网络中的权，也可以用类似邻接矩阵的 $n\times n$ 矩阵表示. 只是此时一条弧所对应的元素不再是 1，而是相应的权而已. 如果网络中每条弧赋有多种权，则可以用多个矩阵表示这些权.

1.3.2 关联矩阵表示法

关联矩阵表示法是将图以关联矩阵(incidence matrix)的形式存储在计算机中. 图 $G=(V,A)$ 的关联矩阵 $\boldsymbol{B}$ 是如下定义的：$\boldsymbol{B}$ 是一个 $n\times m$ 的矩阵，即

$$\boldsymbol{B}=(b_{ik})_{n\times m}\in\{-1,0,1\}^{n\times m},$$

$$b_{ik}=\begin{cases}1, & \exists j\in V, k=(i,j)\in A,\\ -1, & \exists j\in V, k=(j,i)\in A,\\ 0, & \text{其他}.\end{cases}$$

也就是说，在关联矩阵中，每行对应于图的一个节点，每列对应于图的一条弧. 如果一个节点是一条弧的起点，则关联矩阵中对应的元素为 1；如果一个节点是一条弧的终点，则关联矩阵中对应的元素为 -1；如果一个节点与一条弧不关联，则关联矩阵中对应的元素为 0. 对于简单图，关联矩阵每列只含有两个非零元(一个 $+1$，一个 -1). 在关联矩阵中，每行元素 1 的个数正好是对应节点的出度，每行元素 -1 的个数正好是对应节点的入度. 可以看出，这种表示法也非常简单、直接. 但是，在关联矩阵的所有 nm 个元素中，只有 $2m$ 个为非零元. 如果网络比较稀疏，这种表示法也会浪费大量的存储空间. 但由于关联矩阵有许多特别重要的理论性质(我们以后将会介绍)，因此它在网络优化中是非常重要的概念.

例如，图 1.3 所示的网络图，如果关联矩阵中每列对应弧的顺序为(1,2)，(1,3)，(2,4)，(3,2)，(4,3)，(4,5)，(5,3)和(5,4)，则关联矩阵表示为

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0\\ 0 & -1 & 0 & 1 & -1 & 0 & -1 & 0\\ 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1\\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \end{bmatrix}.$$

同样，对于网络中的权，也可以通过对关联矩阵的扩展来表示. 例如，如果网络中每条弧赋有一个权，我们可以把关联矩阵增加一行，把每一条弧所对应的权存储在增加的行中. 如果网络中每条弧赋有多个权，我们可以把关联矩阵增加相应的行数，把每一条弧所对应的权存储在增加的行中.

1.3.3 弧表表示法

弧表表示法将图以弧表(arc list)的形式存储在计算机中. 所谓图的弧表，也就是图的弧集合中的所有有序对. 弧表表示法直接列出所有弧的起点和终点，共需 $2m$ 个存储单元，因此当网络比较稀疏时比较方便. 此外，对于网络图中每条弧上的权，也要对应地用额外的

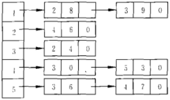
存储单元表示. 例如,图 1.3 所示的网络图,假设弧(1,2),(1,3),(2,4),(3,2),(4,3),(4,5),(5,3)和(5,4)上的权分别为 8,9,6,4,0,3,6 和 7,则弧表表示如下:

起点	1	1	2	3	4	4	5	5
终点	2	3	4	2	3	5	3	4
权	8	9	6	4	0	3	6	7

为了便于检索,一般按照起点、终点的字典序顺序存储弧表,如上面的弧表就是按照这样的顺序存储的.

1.3.4 邻接表表示法

邻接表表示法将图以邻接表(adjacency lists)的形式存储在计算机中. 所谓图的邻接表,也就是图的所有节点的邻接表的集合;而对每个节点,它的邻接表就是它的所有出弧. 邻接表表示法就是对图的每个节点,用一个单向链表列出从该节点出发的所有弧,链表中每个单元对应于一条出弧. 为了记录弧上的权,链表中每个单元除列出弧的另一个端点外,还可以包含弧上的权等作为数据域. 图的整个邻接表可以用一个指针数组表示. 例如,图 1.3 所示的网络图,邻接表表示为



这是一个 5 维指针数组,每一维(上面表示法中的每一行)对应于一个节点的邻接表,如第 1 行对应于第 1 个节点的邻接表(即第 1 个节点的所有出弧). 每个指针单元的第 1 个数据域表示弧的另一个端点(弧的头),后面的数据域表示对应弧上的权. 如第 1 行中的"2"表示弧的另一个端点为 2(即弧为(1,2)),"8"表示对应弧(1,2)上的权为 8;"3"表示弧的另一个端点为 3(即弧为(1,3)),"9"表示对应弧(1,3)上的权为 9. 又如,第 5 行说明节点 5 出发的弧有(5,3)、(5,4),他们对应的权分别为 6 和 7.

对于有向图 $G=(V,A)$,一般用 $A(i)$ 表示节点 i 的邻接表,即节点 i 的所有出弧构成的集合或链表(实际上只需要列出弧的另一个端点,即弧的头). 例如上面例子,$A(1)=\{2,3\}$,$A(5)=\{3,4\}$等. 这种记法我们在本书后面将会经常用到.

1.3.5 星形表示法

星形(star)表示法的思想与邻接表表示法的思想有一定的相似之处. 对每个节点,它也

是记录从该节点出发的所有弧，但它不是采用单向链表而是采用一个单一的数组表示. 也就是说，在该数组中首先存放从节点 1 出发的所有弧，然后接着存放从节点 2 出发的所有弧，依此类推，最后存放从节点 n 出发的所有弧. 对每条弧，要依次存放其起点、终点、权的数值等有关信息. 这实际上相当于对所有弧给出了一个顺序和编号，只是从同一节点出发的弧的顺序可以任意排列. 此外，为了能够快速检索从每个节点出发的所有弧，我们一般还用一个数组记录每个节点出发的弧的起始地址(即弧的编号). 在这种表示法中，可以快速检索从每个节点出发的所有弧，这种星形表示法称为前向星形(forward star)表示法.

例如，在如图 1.3 所示的网络图中，仍然假设弧(1,2)，(1,3)，(2,4)，(3,2)，(4,3)，(4,5)，(5,3)和(5,4)上的权分别为 8,9,6,4,0,3,6 和 7. 此时该网络图可以用前向星形表示法表示如下.

节点对应的出弧的起始地址编号数组(记为 $point$)(在数组 $point$ 中，其元素个数比图的节点数多 1，构造一个虚拟节点 $n+1$)：

节点号 i	1	2	3	4	5	6
起始地址 $point(i)$	1	3	4	5	7	9

记录弧信息的数组：

弧编号	1	2	3	4	5	6	7	8
起点	1	1	2	3	4	4	5	5
终点	2	3	4	2	3	5	3	4
权	8	9	6	4	0	3	6	7

在此表示中，一定有 $point(1)=1$，$point(n+1)=m+1$. 对于节点 i，其对应的出弧存放在弧信息数组的位置区间为$[point(i),\ point(i+1)-1]$. 如果 $point(i)>point(i+1)-1$，则节点 i 没有出弧. 这种表示法与弧表表示法也非常相似，"记录弧信息的数组"实际上相当于有序存放的"弧表". 只是在前向星形表示法中，弧被编号后有序存放，并增加一个数组($point$)记录每个节点出发的弧的起始编号.

前向星形表示法有利于快速检索每个节点的所有出弧，但不能快速检索每个节点的所有入弧. 为了能够快速检索每个节点的所有入弧，可以采用反向星形(reverse star)表示法：首先存放进入节点 1 的所有弧，然后接着存放进入节点 2 的所有弧，依此类推，最后存放进入节点 n 的所有弧. 对每条弧，仍然依次存放其起点、终点、权的数值等有关信息. 同样，为了能够快速检索从每个节点的所有入弧，我们一般还用一个数组记录每个节点的入弧的起始地址(即弧的编号). 例如，图 1.3 所示的网络图，可以用反向星形表示法表示为如下形式.

节点对应的入弧的起始地址编号数组(记为$rpoint$)：

节点号 i	1	2	3	4	5	6
起始地址 $rpoint(i)$	1	1	3	6	8	9

记录弧信息的数组：

弧编号	1	2	3	4	5	6	7	8
终点	2	2	3	3	3	4	4	5
起点	3	1	1	4	5	5	2	4
权	4	8	9	0	6	7	6	3

如果既希望快速检索每个节点的所有出弧，也希望快速检索每个节点的所有入弧，则可以综合采用前向和反向星形表示法. 当然，将弧信息存放两次是没有必要的，可以只用一个数组(*trace*)记录一条弧在两种表示法中的对应关系即可. 例如，可以在采用前向星形表示法的基础上，加上上面介绍的*rpoint*数组和如下的 *trace* 数组即可. 这相当于如下所示的一种紧凑的双向星形表示法.

两种表示法中的弧的对应关系(记为 *trace*)：

反向法中弧编号 j	1	2	3	4	5	6	7	8
正向法中弧编号 $trace(j)$	4	1	2	5	7	8	3	6

对于网络图的表示法，我们作如下说明：

(1)星形表示法和邻接表表示法在实际算法实现中都是经常采用的. 星形表示法的优点是占用的存储空间较少，并且对那些不提供指针类型的语言(如 FORTRAN 语言等)也容易实现. 邻接表表示法对那些提供指针类型的语言(如 C 语言等)是方便的，且增加或删除一条弧所需的计算工作量很少(可以在常数时间内完成)，而这一操作在星形表示法中所需的计算工作量较大(需要花费 $O(m)$的计算时间). 有关“计算时间”的观念是网络优化中需要考虑的一个关键因素，这些概念我们在 1.4 节中再详细介绍.

(2)当网络不是简单图，而是具有平行弧(即多重弧)时，显然此时邻接矩阵表示法是不能采用的. 其他方法则可以很方便地推广到可以处理平行弧的情形.

(3)上述方法可以很方便地推广到处理无向图的情形，但由于无向图中弧没有方向，因此可能需要做一些自然的修改. 例如，可以在计算机中只存储邻接矩阵的一半信息(如上三角部分)，因为此时邻接矩阵是对称矩阵. 无向图的关联矩阵只含有元素 0 和 1，而不含有 −1，因为此时不能区分弧的起点和终点. 又如，在邻接表和星形表示法中，每条弧会被存储两次，而且反向星形表示显然是没有必要的，等等.

1.4 计算复杂性的概念

1.4.1 组合最优化问题

网络优化问题是一类特殊的组合优化问题，因此这里我们首先介绍一些组合优化的基本概念，尤其是它的一些基本性质和特点，即讨论它与我们以前学习过的连续优化(线性和非线性规划)有什么不同.

定义 1.3 所谓组合优化(combinatorial optimization,又称组合最优化)就是离散优化(discrete optimization),它是通过数学方法去寻找离散事件的最优编排、分组、次序或筛选等.这类问题可用数学模型描述为

$$\begin{aligned} &\min\ f(\boldsymbol{x}) \\ &\text{s.t.}\ \ \boldsymbol{g}(\boldsymbol{x})\geqslant \boldsymbol{0}, \\ &\qquad \boldsymbol{x}\in D. \end{aligned}$$

其中,$f(\boldsymbol{x})$为目标函数,$\boldsymbol{g}(\boldsymbol{x})$为约束函数,$\boldsymbol{x}$为决策变量,$D$表示有限个点组成的集合.

一个组合最优化问题可用三个参数(D,F,f)表示,其中D表示决策变量的定义域,F表示可行解区域$F=\{\boldsymbol{x}\mid \boldsymbol{x}\in D,\ \boldsymbol{g}(\boldsymbol{x})\geqslant \boldsymbol{0}\}$,$F$中的任何一个元素称为该问题的可行解,$f$表示目标函数.满足$f(\boldsymbol{x}^*)=\min\{f(\boldsymbol{x})\mid \boldsymbol{x}\in F\}$的可行解$\boldsymbol{x}^*$称为该问题的最优解.组合最优化的特点是可行解集合为有限点集.由直观可知,只要将D中有限个点逐一判别是否满足$\boldsymbol{g}(\boldsymbol{x})\geqslant \boldsymbol{0}$的约束并比较目标值的大小,就可以得到该问题的最优解.因为现实生活中的大量优化问题是从有限个状态中选取最好的,所以大量的实际优化问题是组合最优化问题.例如,第1.1节中介绍的网络优化问题的一些例子都是组合最优化问题.下面再介绍一些例子.

例 1.7 0-1背包问题(knapsack problem)

设有一个容积为b的背包,n个体积分别为$a_i(i=1,2,\cdots,n)$,价值分别为$c_i(i=1,2,\cdots,n)$的物品,如何以最大的价值装包?这个问题称为0-1背包问题.用数学模型表示为

$$\max\ \sum_{i=1}^{n} c_i x_i \tag{1.1}$$

$$\text{s.t.}\ \ \sum_{i=1}^{n} a_i x_i \leqslant b, \tag{1.2}$$

$$x_i \in \{0,1\},\ i=1,2,\cdots,n. \tag{1.3}$$

其中目标函数(1.1)欲使包内所装物品的价值最大,约束条件(1.2)为包的能力限制,约束条件(1.3)表示x_i为二进制变量,$x_i=1$表示装第i个物品,$x_i=0$表示不装.此时,$D=\{0,1\}^n$,F为D中满足约束条件(1.2)的可行解集合,f为目标函数. □

例 1.8 (续例1.6)旅行商问题(TSP)

例1.6中介绍过，TSP问题考虑的是：一个商人欲到n个城市推销商品，每两个城市i和j之间的距离为d_{ij}，如何选择一条道路使得商人每个城市走一遍后回到起点且所走路径最短？考虑到任意两个城市之间的距离可能对称也可能不对称，因此TSP还可以细分为对称TSP和非对称TSP两类．如果$d_{ij}=d_{ji}(\forall i,j)$，此时TSP称为对称距离TSP，否则称为非对称TSP．对一般的TSP(即可能是非对称TSP)，它的一种数学模型描述为

$$\min \sum_{i\neq j} d_{ij}x_{ij} \tag{1.4}$$

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1, i = 1,2,\cdots,n, \tag{1.5}$$

$$\sum_{i=1}^{n} x_{ij} = 1, j = 1,2,\cdots,n, \tag{1.6}$$

$$\sum_{i,j\in s} x_{ij} \leqslant |s|-1, 2\leqslant |s| \leqslant n-2, s\subset\{1,2,\cdots,n\}, \tag{1.7}$$

$$x_{ij} \in \{0,1\}, i,j = 1,2,\cdots,n, i\neq j. \tag{1.8}$$

以上是TSP基于图论的数学模型，其中，约束条件(1.8)表示决策变量$x_{ij}=1$或0，$x_{ij}=1$表示商人行走的路线包含从城市i到城市j的路，而$x_{ij}=0$表示商人没有选择走这条路；$i\neq j$的约束可以减少变量的个数，使得模型中共有$n\times(n-1)$个决策变量．目标函数(1.4)要求商人行走路线的距离之和最小．约束条件(1.5)要求商人只从城市i出来一次，约束条件(1.6)要求商人走入城市j只有一次，因此约束条件(1.5)和约束条件(1.6)表示每个城市经过且只经过一次．但是，仅有约束条件(1.5)和约束条件(1.6)的约束还无法避免圈(或有向圈，有时也称为"回路")的产生．因此，约束条件(1.7)约束旅行商在任何一个城市子集中不形成圈，其中$|s|$表示集合s中元素个数．此时，决策变量的定义域$D=\{0,1\}^{n\times(n-1)}$，可行域F为D中满足约束条件(1.5)，(1.6)，(1.7)和(1.8)的解集合．f为目标函数(1.4)．□

上面两个问题都是组合优化中的经典问题．它们的共性是读者容易理解，可行解集合是有限的，在问题的规模较小时，通过枚举很容易得到最优解．

例1.9　整数线性规划(integer linear programming，ILP)

$$(\text{ILP})\qquad \begin{aligned} &\min\ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ &\text{s.t.}\ \boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}, \\ &\qquad \boldsymbol{x}\geqslant\boldsymbol{0}, \boldsymbol{x}\in\mathbb{Z}^n. \end{aligned}$$

其中，$\boldsymbol{c}$为n维列向量，$\boldsymbol{A}$为$m\times n$矩阵，$\boldsymbol{b}$为m维列向量，$\boldsymbol{x}$为n维决策变量，$\mathbb{Z}$表示整数集合．模型中$\boldsymbol{A}$，$\boldsymbol{b}$和$\boldsymbol{c}$的元素都是整数．□

例1.7和例1.8的数学模型都具有例1.9(ILP)的形式，说明一些组合优化问题可以写成整数线性规划问题．整数线性规划同线性规划在形式上非常相似，不同之处是决策变量全部取整数．

在本书中,我们假设整数线性规划的参数(约束矩阵和右端项系数)都是整数.这主要是因为目前的计算机中都是以有限位储存数据并进行运算.很容易证明,ILP 中的系数是有理数时都可以处理成整数情况.

例 1.10 装箱问题(bin packing)

如何用个数最少的尺寸为 1 的箱子装进 n 个尺寸不超过 1 的物品,该问题为装箱问题. □

例 1.11 约束机器排序问题(capacitated machine scheduling)

n 个加工量为 $\{d_i \mid i=1,2,\cdots,n\}$ 的产品在一台机器上加工,机器在第 t 个时段的工作能力为 c_t,求完成所有产品加工的最少时段数.它的数学模型为

$$\min\ T \tag{1.9}$$

$$\text{s.t.}\ \sum_{t=1}^{T} x_{it} = 1, i = 1,2,\cdots,n, \tag{1.10}$$

$$\sum_{i=1}^{n} d_i x_{it} \leqslant c_t, t = 1,2,\cdots,T, \tag{1.11}$$

$$x_{it} \in \{0,1\}, i = 1,2,\cdots,n; t = 1,2,\cdots,T. \tag{1.12}$$

其中,x_{it} 为决策变量,$x_{it}=1$ 表示第 t 个时段加工产品 i,目标函数(1.9)要求加工所用的时段数最少,约束条件(1.10)表示产品 i 一定在某一个时段加工,约束条件(1.11)表示每个时段的加工量不能超过工作能力的限制. □

有些优化问题可以用整数规划模型的形式表示,如例 1.7,例 1.8 和例 1.11.但有些组合优化问题用整数规划模型表示则比较烦琐和不易被理解,不如原问题的直接叙述简洁和直观,通俗易懂,如例 1.8、例 1.10 和例 1.11.因此,有大量的组合优化问题是通过文字语言叙述的.

1.4.2 多项式时间算法

对于组合优化问题,我们关心的一般不是最优解的存在性和唯一性,而是如何找到有效的算法求得一个最优解.那么如何衡量算法的优劣及有效与无效呢?这就要了解计算复杂性的基本概念.

由组合最优化问题的定义,每一个组合最优化问题都可以通过完全枚举的方法求得最优解.枚举是以时间为代价的,有的枚举时间还可以接受,有的则不可能接受.如例 1.8 的非对称距离 TSP 问题,可以用另一个方法来表示它的可行解:用 n 个城市的一个排列表示商人按这个排列序推销并返回起点.若固定一个城市为起终点,则需要 $(n-1)!$ 个枚举.以计算机 1 s 可以完成 24 个城市所有路径枚举为单位,则 25 个城市的计算时间为:以第 1 个城市为起点,第 2 个到达城市有可能是第 2 个、第 3 个、……、第 25 个城市.决定前两个城市的顺序后,余下是 23 个城市的所有排列,枚举这 23 个城市的排列需要 1 s,所以,25 个城市的枚举需要 24 s.类似地归纳,城市数与计算时间的关系如表 1.1 所示.

表 1.1 枚举时城市数与计算时间的关系

城市数	24	25	26	27	28	29	30	31
计算时间	1 s	24 s	10 min	4.3 h	4.9 天	136.5 天	10.8 年	325 年

通过表 1.1 可以看出，随着城市数的增多，计算时间增加非常之快，当城市数增加到 30 时，计算时间约为 10.8 年，实际计算中已无法接受.

下面我们给出问题及其实例的概念. 问题(problem)是需要回答的一般性提问，通常含有若干个满足一定条件的参数. 问题通过下面的描述给定：①描述所有参数的特性，②描述答案所满足的条件. 问题是一个抽象的模型或概念，它通过一些具体的数据表现出来. 如 TSP 是一个问题，它由例 1.8 的文字或模型(1.4)～(1.8)表示，通过给出的城市数和城市间的距离而使这个问题具体化. 目前计算机可以求解的是这些给出城市数和城市间的距离的具体 TSP. 一个算法常常是针对一个问题来设计的，如上面的 TSP 枚举算法可以求解任何一个具体化的 TSP 的最优解. 而若用计算机求解，则必须对问题中的参数赋予具体值，如 TSP 中的城市数和城市间的距离. 问题中的参数赋予了具体值的例子称为问题的实例(instance). 一个问题通过它的所有实例表现.

衡量一个算法的好坏通常是用算法中的加、减、乘、除和比较等基本运算的总次数同实例在计算机计算时的二进制输入数据的大小关系来度量. 对于一个正整数 x，二进制表示是以

$$x=a_s2^s+a_{s-1}2^{s-1}+\cdots+a_1 2+a_0$$

的系数$(a_sa_{s-1}\cdots a_1a_0)$来表示，其中 $a_s\neq 0, a_i\in\{0,1\}, i=0,1,\cdots,s$. 由

$$2^s\leqslant 2^{\log x}=x\leqslant 2^{s+1}\leqslant 2^{\log x+1},$$

这个二进制数的位数是 $s+1$，也称为数据 x 的输入长度，在 $\log x$ 与 $\log x+1$ 之间. 于是，x 的二进制数的位数是整数$\lceil \log x\rceil$或$\lceil \log x\rceil+1$，但不超过$\lceil \log x\rceil+1$，其中$\lceil x\rceil$表示不小于 x 的最小整数. 特别需要注意的是整数 0 和 1 的二进制位数都是 1，虽说 $\log 0$ 没有意义和 $\log 1=0$. 可以看到，任何一个非负整数的输入长度至少为 1. 在此，在特别定义 $\log 0=0$ 后，上面的二进制表示方法和输入长度的结论可以推广到非负整数. 常规理解的计算机只能用有限位表示一个实数，因此，我们限定只考虑有理数.

如表 1.1 对应的 TSP，当城市数为 n 且第 1 个城市为始终点时，计算一条路径$(1, i_2,\cdots,i_n)$长度的基本运算为两两城市间距离的 n 个求和. 因为有$(n-1)!$条路径，枚举所有路径进行$(n-1)!$次比较，可以得到最优路径. 这个枚举算法的基本计算总次数为$(n-1)!\cdot n=n!$. 实例在计算机计算时只需输入城市数和所有城市之间的距离. 于是，输入数据是城市数 n 和城市间的距离 $S=\{d_{ij}\mid 1\leqslant i,j\leqslant n, i\neq j\}$，且假设这些数为整数. 对任何一个非零距离 d_{ij}，它的二进制输入数据的输入长度不超过 $\log|d_{ij}|+1$. 实例的二进制输入长度总量不超过

$$L=n(n-1)+\log|P|,$$

其中，$P=\prod\{d_{ij}\mid d_{ij}\neq 0\}$.

若假设 $S=\{d_{ij}\mid 1\leqslant i,j\leqslant n,i\neq j\}$ 中的每一个数据的绝对值都有上界 K，则有 $L=n(n-1)+\log|P|\leqslant n(n-1)(1+\log K)$. 此时发现，TSP 实例的计算机二进制输入长度和算法的基本计算总次数都是城市数 n 的函数. TSP 实例的二进制输入长度是 n 的多项式函数. TSP 枚举算法所需的基本计算的总次数总是 n 的阶乘函数，比指数函数增加的速度还快.

我们对实例的二进制输入长度和算法的基本计算总次数一般是进行粗略估计，通常只是给出一个上限. 一个求解实例 I 的算法的基本计算总次数 $C(I)$ 同实例 I 的计算机二进制输入长度 $d(I)$ 的关系常用符号 $C(I)=f(d(I))=O(g(d(I)))$ 表示，它的含义是：求解实例 I 的算法的基本计算总次数 $C(I)$ 是实例输入长度 $d(I)$ 的一个函数，这个函数被另一个函数 $g(x)$ 控制，即存在一个函数 $g(x)$ 和一个非负常数 α，使得

$$C(I)\leqslant \alpha g(d(I)). \tag{1.13}$$

此式中 $g(x)$ 的函数特性决定了基本计算总次数的性能.

定义 1.4 假设问题和解决该问题的一个算法已经给定，若给定该问题的一个实例 I，存在多项式函数 $g(x)$，使得式(1.13)成立，我们称该算法对实例 I 是多项式时间算法 (polynomial time algorithm)；若存在 $g(x)$ 为多项式函数且对该问题任意的一个实例 I，都有式(1.13)成立，则称该算法为解决该问题的多项式时间算法. 特别对优化问题，若一个算法为求解该优化问题最优解的多项式时间算法，则称其为该优化问题的多项式时间最优算法. 多项式时间算法也常常简称为多项式算法.

为了方便，当不存在多项式函数 $g(x)$ 使得式(1.13)成立时，称相应的算法是非多项式时间算法，或指数时间算法(exponential time algorithm)，或简称为指数算法. 相比较而言，随着变量的增加，多项式函数增长的速度比指数函数增长的速度要慢得多，例如，随着 n 的增加，n^k(k 为正整数)的增长速度远比 a^n($a>1$ 为实数)增长的速度慢. 因此，我们更喜欢多项式函数. 若一个多项式时间算法的基本计算总次数为 $C_1=2^{10}n^2$，而另一个非多项式时间算法的计算次数为 $C_2=2^n$. 当 n 比较小的时候，多项式时间算法的优越性并不明显. 如当 $n=10$ 时，$C_1/C_2=10^2$，多项式时间算法的计算次数是非多项式时间算法的 10^2 倍. 但当$n=2^5$ 时，$C_1/C_2=2^{-12}$，非多项式时间算法的计算次数上升非常快，是多项式时间算法的计算次数的 2^{12}倍.

由上面的讨论看出，在实例的二进制输入规模增大时，多项式时间算法的基本计算总次数的增加速度相对较慢. 目前习惯将算法分为多项式时间算法和指数时间算法，且多项式时间算法一般也通俗地称为有效算法或好算法等，而指数时间算法则被认为是无效算法或坏算法等. 这种归类方法主要考虑的是 n 增加时的变化趋势. 重述一个观点，我们构造算法的目的是能够解决问题的所有实例而不单单是问题的一个实例. 根据这个观点，再结合问题、实例和算法，形成了复杂性分析的概念.

复杂性分析的一个研究方向是对算法进行评价. 对于解决一个问题的算法,如何评估这个算法的性能? 复杂性分析是通过式(1.13)在最坏实例的条件下,确定是否存在多项式函数 $g(x)$,即可以叙述为:对一个求解问题的算法,是否存在多项式函数 $g(x)$ 和非负常数 α,使得对问题的任意一个实例 I,都有式(1.13)成立? 这种分析方法称为最坏情况分析(worst-case analysis).

我们本书后面各章主要讨论网络优化问题的最优算法. 在对网络优化算法进行复杂性分析时,首先也应该确定实例的输入规模(长度). 从第1.3节可以看出,在计算机上采用不同数据结构表示网络(图)时,输入规模是不同的. 值得注意的是,在大量的研究文献中,当分析算法的复杂度时,为了避免陷入对输入规模的细节进行讨论,一般常用的方法是用优化问题的一些具有代表性的参数来作为输入规模. 我们在本书的后续章节中也采用这种广为接受的方法,即用优化问题的一些具有代表性的参数来作为输入规模. 通常采用的是图的节点个数 n 和弧的条数 m,以及问题参数(如弧上的权或节点上的权等)的最大值 θ(按绝对值)等作为网络优化问题的实例的输入规模. 可以看出,实际问题的输入规模一定是 m,n 和 $\log\theta$ 的一个多项式函数(为什么? 请读者思考). 所以,如果网络优化算法是多项式时间算法,则运行时间的上界一定是 m,n 和 $\log\theta$ 的多项式函数. 特别地,如果运行时间的上界是 m,n 的多项式函数(即该多项式函数不包含 $\log\theta$),则称相应的算法为强多项式(strongly polynomial)时间算法. 也就是说,强多项式(时间)算法的运行时间的上界不依赖于问题参数的最大值 θ,它是一种特殊的多项式算法. 我们本书后面将要介绍的网络优化算法中有许多是强多项式算法. 但是,如果算法运行时间的上界是 m,n 和 θ 的多项式函数,则称相应的算法为伪多项式(pseudopolynomial)时间算法. 也就是说,如果可能采用"1进制"编码,伪多项式算法的运行时间的上界一定是输入规模的多项式函数. 所谓"1进制"编码,这里是指对问题实例中的一个具体数值(非负整数)x,假设其编码长度就是 x,而不是通常 β 进制编码下的输入长度 $O(\log_\beta x)$($\beta>1$ 为整数). 由于采用"1进制"编码在目前的计算机上是不现实的,所以一般来说,伪多项式(时间)算法并不是真正的多项式算法. 例如,我们以后将可以看到,例1.7介绍的背包问题不存在多项式算法,但却存在伪多项式算法.

1.4.3 多项式问题

除了对算法进行最坏性能评价以外,复杂性分析的另一个研究方向是对组合优化问题归类. 多项式问题可以作如下定义.

定义 1.5 对于给定的一个优化问题,若存在一个求解该问题最优解的算法、一个多项式函数 $g(x)$ 和非负常数 α,使得式(1.13)对给定优化问题的任何一个实例成立,则称给定的优化问题是多项式时间可解问题,或简称多项式问题,所有多项式问题集记为 P(polynomial).

同样道理,我们也可以定义强多项式问题、伪多项式问题等.

例 1.12 线性规划问题(linear programming,LP)

$$
\text{(LP)}\quad \begin{aligned} &\min\ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ &\text{s. t. } \boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}, \\ &\qquad \boldsymbol{x}\geqslant \boldsymbol{0}, \boldsymbol{x}\in \mathbb{R}^n. \end{aligned}
$$

其中，$\boldsymbol{c}$ 为 n 维列向量，$\boldsymbol{A}$ 为 $m\times n$ 矩阵，$\boldsymbol{b}$ 为 m 维列向量，$\boldsymbol{A},\boldsymbol{b},\boldsymbol{c}$ 各分量为整数(分量为有理数等价于整数)，$\boldsymbol{x}$ 为 n 维决策变量. □

单纯形算法是解决线性规划问题的主要算法之一，对于大多数实例，它的计算效果非常好，这可从实际工程、管理人员的大量实际应用中得以证实. 也可以说，对线性规划问题的很多实例而言，单纯形算法的运算时间(操作次数)是可以被输入长度(规模)的多项式控制的. 但也存在极端的情形，例如早在 1971 年就有学者给出过下面这样一个实例：

$$
\begin{aligned} &\min\ -x_n \\ &\text{s. t. } 4x_1-4r_1=1 \\ &\qquad x_1+s_1=1, \\ &\qquad 4x_j-x_{j-1}-4r_j=0,\ j=2,3,\cdots,n, \\ &\qquad 4x_j+x_{j-1}+4s_j=4,\ j=2,3,\cdots,n, \\ &\qquad x_j,r_j,s_j\geqslant 0,\ j=1,2,\cdots,n. \end{aligned} \tag{1.14}
$$

可以证明用单纯形算法求解线性规划(1.14)，其迭代步数是 2^n-1. 由此可见，对线性规划的单纯形算法，不存在多项式函数 $g(x)$ 和非负常数 α，使得对问题的任意一个实例 I，都有式(1.13)成立. 也就是说单纯形算法不是线性规划问题的多项式时间算法，但这并不能说明线性规划问题不属于多项式问题. 前苏联科学家 Khachian 在 1979 年成功地构造了线性规划问题的椭球算法并证明了该算法是多项式时间算法，而且许多学者又在 20 世纪 80 年代提出了求解线性规划问题的许多种内点算法，这些算法也是多项式算法. 因此，线性规划问题是多项式问题.

本书后面章节将要介绍的许多网络优化问题属于一类特殊的组合优化问题，即可以找到多项式时间算法的问题. 对于这样的多项式问题，理论上我们主要关心的是找到计算工作量尽可能小的算法，即复杂度尽可能低的算法；而从实际应用的角度，我们还关心算法的实用效果，希望找到实际计算效率很高的算法.

并不是所有的组合优化问题都找到了求解最优解的多项式时间算法. 也就是说，还不能肯定一些组合优化问题是否属于 P. 经过几代数学家的努力，他们研究、整理了一类难以求解的组合最优化问题，迄今为止，这些问题还没有一个有能求得最优解的多项式时间算法. 这一类组合最优化问题归为所谓的 NP-hard 问题. 受人类认识能力的限制，目前人们只能假设这一类难的组合最优化问题不存在求解最优解的多项式时间算法. 本节中例 1.7、例 1.8、例 1.9、例 1.10 和例 1.11 的组合最优化问题都属于这类难解的问题. 有关复杂性的一些更多的严格的概念和一个问题复杂性的严格确定将在第 2 章进一步讨论. 感兴趣的读者可以继续阅读 2.1 节或参考计算复杂性的其他专著(如文献[6]).

练 习 题

1. 举例说明什么是网络优化问题？

2. 证明：在任何两人或两人以上的人群中，至少有两个人他们的朋友一样多.

3. 证明：一个无向图是二部图的充要条件是它不含奇圈(即长度为奇数的圈).

4. 设 $\boldsymbol{C}$ 为图 G 的邻接矩阵，$\boldsymbol{B}$ 为图 G 的关联矩阵，问 $\boldsymbol{C}$ 与 $\boldsymbol{BB}^{\mathrm{T}}$ 之间有什么关系？分别对 G 为有向图和无向图的情况进行讨论.

5. 证明：n 阶连通有向图的关联矩阵的秩为 $n-1$. 这一结论对无向图是否成立？

6. 试分析网络图在计算机上的各种表示方法的优缺点.

7. 用本章介绍的各种表示方法在计算机上表示图 1.4 的有向网络图. 去掉图中弧的方向后，用本章介绍的各种表示方法在计算机上表示该无向网络图.

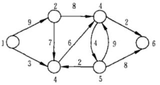


图 1.4 第 7 题图

8. 举例说明什么是组合最优化问题？并以此说明问题与实例的区别.

9. 请你设计一个复杂度尽可能低的算法，将 n 个自然数由小到大排列起来，并分析其计算复杂度.

10. 给出求解下列每一个问题的算法. 在每一个问题里固定输入的表示，即给出它的一个例子，把你的算法所需时间的上界表示为输入长度的函数.

(1) 给定平面上 $n(>2)$ 条直线，$\{a_ix+b_iy=c_i \mid i=1,2,\cdots,n\}$，其中 $\{a_i,b_i,c_i \mid i=1,2,\cdots,n\}$ 为整数. 这些线有公共点吗？

(2) 给定一个整数 p，p 是素数吗？

(3) 给定 n 个整数，选出最大数.

11. 设 $f(n)$，$g(n)$ 是定义在正整数上的正实值函数，如果存在一个正常数 α，使得当 n 足够大时有 $f(n)\leqslant \alpha g(n)$，则记 $f(n)=O(g(n))$. 证明：对一切给定的正整数 k 和 $\varepsilon>0$，有 $(\log n)^k=O(n^{\varepsilon})$，$n^{\log n}=O(1+\varepsilon)^n$.

12. f 和 g 的定义如下：

$$f(n)=\begin{cases} n^2, & 若 n 为素数, \\ n^3, & 其余, \end{cases} \qquad g(n)=\begin{cases} n^2, & 若 n 为奇数, \\ n^3, & 其余. \end{cases}$$

下述哪个结论对？

(1) $f(n)=O(g(n))$，

(2) $g(n)=O(f(n))$，

(3) $f(n),g(n)=O(n^1)$，

(4) $f(n),g(n)=O(n^2)$.

第2章　算法基础

在介绍具体的网络优化问题与算法之前，我们需要了解一些算法的基本理论和方法. 本书所介绍的网络优化问题大多是多项式时间可解问题，但是还有大量的组合优化问题目前还没有找到多项式时间算法. 当我们面对一个网络优化问题，或者组合优化问题时，在设计与分析算法之前，首先要判断这个问题是否为多项式可解，这就涉及算法复杂性理论. 算法复杂性理论是算法领域的理论基石，本章将对它进行简要的介绍. 本章还将介绍在算法设计中与分析中一些常用的方法，如贪婪算法（greedy algorithm）、动态规划（dynamic programming）、线性规划、二分法等. 最后，给出一个简单的网络优化算法.

2.1　NP，NPC和NP-hard概念

1.4节初步介绍了计算复杂性的一些概念，我们知道了多项式问题，同时也知道像例1.7～例1.11的组合优化问题还没有找到多项式时间的最优算法. 本节将以组合最优化问题为研究对象，通过用几个典型组合优化问题一步一步地展开、深入，逐渐理解有关NP、NPC和NP-hard的概念. 同时，介绍如何分析一个组合优化问题的计算复杂性. 计算复杂性问题的研究涉及图灵（Turing）机概念. 在此，仅用组合优化中较为直观和通用的语言简单描述.

2.1.1　问题、实例与输入规模

首先，我们介绍一些有关问题复杂性的基本概念. 复杂性的研究是从区分“问题”和“实例”并定义实例的“输入规模”开始的，我们在第1章对此已经作了一些介绍，下面再简单回顾一下这些概念.

给定一个组合优化问题，当问题中的参数赋予具体值时，称为问题的一个实例. 这些具体值称为数据，这些数据输入计算机所占的空间称为实例的输入长度（input length）或输入规模（input size）.

也就是说，一旦问题中所有的参数确定具体值以后，我们称之为问题的一个实例，且实例的输入长度同计算机中的存储编码（encoding）方式有关. 对问题、实例和输入长度3个概念，我们用表2.1和表2.2所列内容来理解（表2.2中假设采用二进制编码）.

在1.4节说过，以β进制（$\beta>1$且为正整数）记录一个正整数n，即若n的展开形式为

$$n=a_k\beta^k+a_{k-1}\beta^{k-1}+\cdots+a_0\beta^0,$$

其中$a_k\neq0$，$a_i\in\{0,1,\cdots,\beta-1\}$，$i=0,1,\cdots,k$，则它的$\beta$进制数据为$a_ka_{k-1}\cdots a_0$. 所以输入长

度 $k+1$ 在 $\log_\beta n$ 与 $\log_\beta n+1$ 之间，或用整数 $\lceil \log_\beta n \rceil+1$ 估计输入长度的上限. 对包含多个数据的实例，采用数据累计的方法记录实例的输入长度. 由于计算机中只有有限位有效数字，故所有数据都以有理数讨论.

注：此处都是对非负整数考虑输入长度，即没有考虑实际数据中的负号. 它们将在下面讨论的算法中体现.

表 2.1　问题和实例

问　题	实　例
TSP	例 1.8 问题中的各参数为：100 个城市，城市间距离 d_{ij} 已知.
背包问题	例 1.7 问题中的各个参数为：4 个物品，大小分别为 4,3,2,2，价值分别为 8,7,5,7. 包的大小为 6.
整数线性规划	例 1.9 问题中的 n, $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$ 已知.

表 2.2　编码与输入长度

数据	二进制编码	输入长度
1	1	1
2	10	2
10	1010	4
64	1 000 000	7
$k\geqslant 2$		$\leqslant \lceil \log k \rceil+1$

一个实例完全由它的数据决定. 给定一个问题的数据，即给定一个实例，我们可以用一种已知的计算机上的方法去求解. 这种计算机上的求解方法称为算法. 算法不仅仅局限于每一个实例，而是求解问题的通用程序. 评价算法的一个主要指标是计算所耗用的时间. 耗用的时间可以通过两种方法计量. 第一种计量方法是计算机中央处理器处理实例的工作时间. 由于计算时间可能因计算机的运算速度不同而产生差别，这种计量方法因计算机的不同而失去了比较的效能. 另一种计量方法是估计算法的基本运算总次数，即算法中加、减、乘、除和比较等的总次数. 无论如何，算法的加、减、乘、除和比较等基本运算是确定的，于是，一个算法基本运算的总次数也就称为该算法的计算时间. 衡量一个算法的计算效率应该考虑计算时间和其计算实例的输入长度. 一个算法解决一个问题，我们以此为目的构造算法. 从 1.4 节中的介绍我们可以知道：一个算法的功效是通过该算法的最坏实例的计算时间来评价；一个问题是否为多项式问题是通过研究是否存在一个最优算法和一个多项式函数 $g(x)$ 使得式(1.13)对所有的实例成立.

迄今为止，对许多的组合优化问题都没有找到求最优解的多项式时间算法，因此，比多项式问题类可能更广泛的一个问题类是非确定多项式 (nondeterministic polynomial, NP)

问题. NP 的概念是由判定问题引入的.

2.1.2 判定问题

定义 2.1 如果一个问题的每一个实例只有"是"或"否"两种答案,则称这个问题为判定问题(decision problem). 称有肯定答案的实例为"是"实例. 称答案为"否"的实例为"否"实例或非"是"实例.

在研究组合优化问题复杂性时,处理的方法是对给定的一类优化问题,将其转化为判定问题,使对每一个实例只有"是"或"否"的回答. 例 2.1 和例 2.2 给出几个优化问题对应的判定问题.

例 2.1 例 1.12 的线性规划问题(LP)转化为 LP 判定问题

先给出一个目标值 z,将极小化目标函数转化为:判定是否有可行解使其目标值不超过这个 z. 判定问题的数学表达式为:给定 z,是否有

$$\{\boldsymbol{x} \mid \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \leqslant z, \boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}, \boldsymbol{x} \geqslant \boldsymbol{0}\} \neq \varnothing ?$$

其中 $\boldsymbol{x} \in \{\boldsymbol{x} \mid \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \leqslant z, \boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}, \boldsymbol{x} \geqslant \boldsymbol{0}\}$ 称为对应线性规划判定问题的可行解. 同样,整数线性规划也可以写成同以上类似的判定问题. □

例 2.2 例 1.8 的 TSP 问题对应的判定问题

用 n 个城市的一个排列 $i_1 i_2 \cdots i_n$ 表示商人从城市 i_1 出发依次通过 $i_2, \cdots, i_n$,最后返回 i_1 这样一个路径. 判定问题为:给定 z,是否存在 n 个城市的一个排列 $W = i_1 i_2 \cdots i_n$,使得

$$f(W) = \sum_{j=1}^{n} d_{i_j i_{j+1}} \leqslant z ?$$

其中 $i_{n+1} = i_1$. 满足 $f(W) = \sum_{j=1}^{n} d_{i_j i_{j-1}} \leqslant z$ 的一个排列 W 称为对应判定问题的一个可行解. □

如下面的例 2.3 和例 2.4 所示,有些组合问题本身就是一个判定问题.

例 2.3 适定性问题(satisfiability problem)

在逻辑运算中,布尔变量 x 的取值只有两个:"真"和"假",逻辑运算有"或(+)","与(·)"和"非($\overline{\ }$)". 如果"真"用"1"表示,"假"用"0"表示,布尔运算遵循表 2.3(a),(b),(c)的运算规律.

表 2.3(a) 运算"+"

布尔变量值		运算结果
0	0	0
0	1	1
1	0	1
1	1	1

表 2.3(b) 运算"·"

布尔变量值		运算结果
0	0	0
0	1	0
1	0	0
1	1	1

表 2.3(c) 运算"$\overline{\ }$"

布尔变量值	运算结果
0	1
1	0

设$\{x_1,x_2,\cdots,x_n\}$为n个布尔变量，布尔变量及其"非"组成的集合$\{x_1,x_2,\cdots,x_n;\bar{x}_1,\bar{x}_2,\cdots,\bar{x}_n\}$称为文字集. 文字集的任意一个子集及其各元素的"或"运算组成一个句子. m个句子及它们之间的"与"运算组成一个表达式. 如表达式

$$(x_1+\bar{x}_2)\cdot x_2$$

的两个句子为$x_1+\bar{x}_2$和x_2. 当每一个逻辑变量取一定值时，采用逻辑运算关系，可以计算布尔表达式的值. 在上式中，当$x_1=1,x_2=1$时，表达式为"真". 一组使表达式为"真"的变量取值称为真值分配. 存在真值分配的表达式称为适定的. 容易证明，表达式

$$(x_1+\bar{x}_2)\cdot(\bar{x}_1+x_2)\cdot(x_1+x_2)\cdot(\bar{x}_1+\bar{x}_2)$$

就不是适定的.

适定性问题定义为：

给定包含n个布尔变量的m个句子$C_1,C_2,\cdots,C_m$，布尔表达式$C_1\cdot C_2\cdot\cdots\cdot C_m$是适定的吗? □

例 2.4 三精确覆盖(3-exact covering)

已知$S=\{u_1,u_2,\cdots,u_{3m}\}$的$n$个子集构成的子集族$F=\{S_1,S_2,\cdots,S_n\}$，其中每个子集包含$S$中3个元素，$F$中是否存在$m$个子集$S_{i_1},S_{i_2},\cdots,S_{i_m}$，使得$\bigcup_{j=1}^{m}S_{i_j}\supseteq S$? 若$m$个子集$S_{i_1},S_{i_2},\cdots,S_{i_m}$满足$\bigcup_{j=1}^{m}S_{i_j}\supseteq S$，则称$m$个子集$S_{i_1},S_{i_2},\cdots,S_{i_m}$覆盖$S$. □

2.1.3 非确定多项式问题类(NP)

对于判定问题，我们可以将求解该问题的算法分为两个阶段. 第1阶段称为求解或猜测阶段. 它的主要工作是求解或猜测该问题的一个解. 对于"是"实例，我们的目标是能够找到回答"是"实例的解. 第2个阶段称为检查或验证阶段. 一旦解已经选定，将求出的解或猜测的解作为输入，验证此解是否为该实例"是"的回答. 我们称实例"是"回答的解为实例的可行解，否则称为不可行解. 求解一个实例是否存在回答"是"的解是难度较大的工作，对于很多优化问题，目前不能在多项式时间内完成. 因此，我们考虑降低求解的要求，定义一类相对"容易"的问题：对于一个判定问题的任一个"是"实例，是否可以在多项式时间内验证它确实是"是"实例? 这就引出了一类更广泛的问题：非确定多项式问题类(NP).

在计算机计算的过程中，问题的每一个实例在计算机输入时都以一定结构的字符串表示，如例1.8的TSP问题的每一个实例可以用一个长度不超过$n(n-1)+\log|P|$的0-1字符串表示，其中$P=\Pi\{d_{ij}\mid d_{ij}\neq 0\}$. 时常采用"解"表示问题的一个解决方案. 实例的每一个解也同样可以用一个字符串表示.

定义 2.2 若存在一个多项式函数$g(x)$和一个验证算法H，使得：判定问题A的任何一个实例I为"是"实例当且仅当存在一个验证字符串S，满足其输入长度$d(S)$不超过$g(d(I))$，其中$d(I)$为I的输入长度，且算法H验证实例I为"是"实例的计算时间$f(H)$不超过$g(d(I))$，则称判定问题A是非确定多项式的，简记为NP. 在不混淆的情况下，用NP表示所有非确定多项式问题的集合.

由上面的定义知,判定问题是否属于 NP 的关键是对"是"判定实例,"存在"一个验证字符串和算法,满足以上定义.我们不须回答"存在"的算法和字符串是怎样得到的,这里验证字符串的含义一般可以理解为实例的一个解.定义 2.2 中两处用到多项式的限制,一处是"验证"字符串 S 的输入长度不超过 $g(d(I))$,另一处是算法 H 验证实例 I 为"是"实例的计算时间不超过 $g(d(I))$.可以将上面定义中这两处的多项式限制改变为:存在两个多项式函数 $g_1(x)$ 和 $g_2(x)$,使得"验证"字符串 S 的输入长度不超过 $g_1(d(I))$,算法 H 验证实例 I 为"是"实例的计算时间不超过 $g_2(d(I))$.因为只要选高阶的多项式作为定义 2.2 中的 $g(x)$,这样的定义与原定义 2.2 等价.

对非"是"实例,我们不讨论它的相关判定的结果.由定义可知 P⊆NP.用下面例 2.5、例 2.6 理解如何确定一个判定问题是否属于 NP.

例 2.5(续例 2.2) TSP 问题属于 NP

例 2.2 给出了对应的判定问题.对任何一个 n 城市 TSP 实例,它的解形式是$\{1,2,\cdots,n\}$的一个排列 $W=i_1i_2\cdots i_n$,显然该字符串输入长度不超过 $\sum_{i=1}^{n}(\lceil\log i\rceil+1)<n+n\lceil\log n\rceil$.实例的输入长度可由城市间的距离数据确定,城市间的距离数据有 $n(n-1)$ 个(不考虑城市自身间的距离),城市 i 和 j 之间的一个非零距离 d_{ij} 的输入长度不低于 $\max\{1,\log|d_{ij}|\}$,因此,实例的输入长度不低于 $L=\max\{n(n-1),\log|P|\}$,其中,$P=\prod\{d_{ij}\mid d_{ij}\neq 0\}$.这样估计出实例的输入长度不低于 $n(n-1)$.上面的讨论表明,TSP 任何一个解的输入长度不超过 $n(1+\lceil\log n\rceil)\leqslant n(n-1)+4$,不超过实例输入长度 $L+4$.

若给定 TSP 判定实例的一个可行解,只需验证

$$f(W)=\sum_{j=1}^{n}d_{i_ji_{j+1}}\leqslant z?$$

验证算法有 n 个加法和 1 个比较,算法的计算时间为 $n+1<L+4$.于是,定义 2.2 中的多项式函数可选 $g(x)=x+4$,验证算法的计算时间和判定问题实例可行解的输入长度都满足定义 2.2 的要求,故 TSP∈NP. □

例 2.6(续例 2.4) 三精确覆盖属于 NP

集合 S 中共有 $3m$ 个元素,按下标排列这些元素,此时,子集族 F 中的每一个元素 $S_i(i=1,2,\cdots,n)$对应一个 $3m$ 维向量.向量形式为:向量的 $3m$ 个分量分别对应 $3m$ 个元素,包含在对应子集中的元素为 1,余下的为 0.例如,6 个元素 $S=\{u_1,u_2,\cdots,u_6\}$,F 中的一个元素(即 S 的一个子集)$S_1=\{u_1,u_2,u_6\}$,对应向量为$\boldsymbol{\alpha}_1=(1,1,0,0,0,1)$.表示为一个字符串(110001).$F$ 中 m 个元素$\{S_{i_1},S_{i_2},\cdots,S_{i_m}\}$是 F 的一个精确三覆盖的充分必要条件为字符串向量问题满足:相应 m 个字符串对应的向量 $\boldsymbol{\alpha}_{i_1},\boldsymbol{\alpha}_{i_2},\cdots,\boldsymbol{\alpha}_{i_m}$ 满足 $\sum_{j=1}^{m}\boldsymbol{\alpha}_{i_j}=(1,1,\cdots,1)$.

无论如何,三精确覆盖问题任何一个实例的输入长度至少需要占用长度 $3m$.上面的讨论得到,三精确覆盖问题转化为一个字符串向量问题.按字符串向量问题,三精确覆盖任何

一个"是"实例的解可以用输入长度为 $3m^2$ 的字符表示. 验证 $\sum_{j=1}^{m} \boldsymbol{\alpha}_{i_j}=(1,1,\cdots,1)$ 是否成立的算法需 $3m^2$ 个加法和 $3m$ 个比较,算法的计算时间为 $3m^2+3m$. 多项式函数可选 $g(x)=3x^2+x$,则定义 2.2 条件满足,所以三精确覆盖属于 NP. □

研究中我们常常有这样一种思路,设法将几类问题归结为一个问题,一旦解决了这一个归结后的问题,其他的几类问题也就得到解决. 我们称这种方法为归类或归约. 处理判定问题也有类似的方法.

定义 2.3 给定问题 A1 和 A2,若存在一个多项式函数 $g(x)$和一个多项式算法 φ 满足:

(1) 对 A1 的任何一个实例 I_1,在实例 I_1 的输入长度 $d(I_1)$的多项式时间 $g(d(I_1))$内,算法 φ 将实例 I_1 的输入转换为 A2 的一个实例 I_2 的输入;

(2) 算法 φ 的这种转换(构造)过程使得实例 I_1 和 I_2 的解一一对应,即 φ 在多项式时间内将实例 I_1 的一个解 d_1 的输入转换为实例 I_2 的一个解的输入,且 d_1 为 I_1 的"是"答案的充分必要条件是 d_1 对应的解是 I_2 的一个"是"答案.

则称问题 A1 多项式转换(transformation)为问题 A2,算法 φ 称为问题 A1 到问题 A2 的一个转换算法.

可见,此时的多项式算法 φ 是一种字符串映射关系,即将 A1 的实例 I_1 的输入字符串转换为 A2 的一个实例 I_2 的输入字符串,将实例 I_1 的一个解 d_1 的输入字符串转换为实例 I_2 的一个解的输入字符串. 下面我们通过例 2.7 和例 2.8 来理解定义 2.3.

例 2.7(续例 2.3) 适定性问题多项式转换为整数线性规划问题.

将适定性问题的逻辑变量"真"对应"1","假"对应"0",于是逻辑变量 x 对应整数变量,$\bar{x}$ 对应 $1-x$. 一个 n 个变量的句子 $C_j(j=1,2,\cdots,m)$要求是"真",对应下列不等式是否有可行解:

$$\sum_{x_i \in C_j} x_i + \sum_{\bar{x}_i \in C_j} (1-x_i) \geqslant 1, \quad j=1,2,\cdots,m, \tag{2.1}$$

$$x_i \in \{0,1\}, \quad i=1,2,\cdots,n. \tag{2.2}$$

对于适定性问题的任何一个实例,它是由 n 个逻辑变量和 m 个句子组成. 按实例的定义,m 个句子中的逻辑变量值是已知的数据,故适定性问题实例的输入长度是 mn. 由式(2.1)和式(2.2)的映射,对应 m 组式(2.1)和 n 组式(2.2)形式的不等式是整数线性规划问题,不难算出对应整数线性规划实例的输入长度不超过 $2mn+m\log n$. 因此,定义 2.3 中的多项式函数选 $g(x)=3x^2$,就有定义 2.3 的(1)成立. 易证适定性问题到对应式(2.1)和式(2.2)约束的整数规划问题满足定义 2.3 的(2),说明适定性问题多项式转换为整数线性规划问题. □

研究例 1.7 的 0-1 背包问题的一种特殊情形 $c_j=a_j, j=1,2,\cdots,n$,我们可以给出一个特殊 0-1 背包问题: 子集和问题(subset sum problem). 我们可以给出它的一个判定问

题为：

对给定的整数 $c_j(j=1,2,\cdots,n)$ 和 b，是否存在 $\{1,2,\cdots,n\}$ 的子集 B，使得 $\sum_{j\in B} c_j = b$?

例 2.8 三精确覆盖多项式转换为子集和判定问题.

由例 2.6 的 S_j 与二进制向量的对应关系，一个二进制向量又可与一个整数一一对应：

$$S_j \to c_j = \sum_{u_i \in S_j} (n+1)^{i-1} \tag{2.3}$$

如此，将一个向量一一对应于一个整数. 对于 $3m$ 个分量全为 1 的向量 $(1,1,\cdots,1)$，选 b 为

$$(1,1,\cdots,1) \to b = \sum_{j=1}^{3m} (n+1)^{j-1}. \tag{2.4}$$

容易验证上面对应满足定义 2.3 的(1). 由上面的构造，三精确覆盖实例的一个解 $\{S_{i_1}, S_{i_2},\cdots,S_{i_m}\} \subseteq F$ 一一对应子集和判定问题实例的一个解 $B=\{i_1,i_2,\cdots,i_m\}$. 此时，S 存在一个精确三覆盖 $\{S_{i'_1}, S_{i'_2},\cdots,S_{i'_m}\}$ 的充要条件为 $\sum_{i=1}^{m} c_{i'_i} = b$.

充分性：假设存在 $T\subseteq\{1,2,\cdots,n\}$，使得 $\sum_{j\in T} c_j = b$. 由式(2.3)和式(2.4)，b 的每个指数项系数为 1. 又因为 T 的总项数不超过 n，其同次幂项的系数和不超过 n，因而，由 T 和式(2.4)，$n+1$ 的每个次幂项系数恰好为 1. 由式(2.3)的反变换，得 $S_j, j\in T$ 是 S 的一个三精确覆盖.

必要性：由式(2.3)和式(2.4)的定义，可得结论.

由上面的讨论，映射关系(2.3)和(2.4)满足定义 2.3 的(2)，所以，三精确覆盖多项式时间转换为子集和判定问题. □

2.1.4 NP 完全问题类(NPC)

Cook 在 1971 年给出并证明了有一类问题具有下述性质：

(1) 这类问题中任何一个问题至今未找到多项式时间算法；

(2) 如果这类问题中存在一个问题有多项式时间算法，那么这类问题都有多项式时间算法.

这类问题就是所谓的 NP 完全问题. 下面逐步给出其严格定义.

定义 2.4 已知判定问题 A1 和 A2，若存在多项式函数 $g_1(x)$ 和 $g_2(x)$，使得对 A1 的任何实例 I，在多项式时间内构造 A2 的一个实例，其输入长度不超过 $g_1(d(I))$，并对 A2 的任何一个算法 $H2$，都存在问题 A1 的一个算法 $H1$，使得 $H1$ 算法调用 $H2$ 算法并使得计算时间 $f_{H1}(d(I)) \leqslant g_2(f_{H2}(g_1(d(I))))$，其中 $f_{H1}(d(I))$ 和 $f_{H2}(d(I))$ 分别表示算法 $H1$ 和 $H2$ 的计算时间与实例输入长度 $d(I)$ 之间的关系，则称问题 A1 多项式归约(reduction)为 A2 问题.

从定义 2.4 可以看出，若 A2 存在多项式时间算法，则 A1 一定存在多项式时间算法，因此，问题 A2 的难度不低于问题 A1. 再比较定义 2.3 和定义 2.4，若问题 A1 多项式转换为

问题 A2，对 A2 的一个算法 $H2$，构造求解 A1 问题的算法可遵循如下步骤：对问题 A1 的任何一个实例 I_1，先用多项式时间 $g_1(d(I_1))$ 构造 A2 问题的一个实例 I_2；再调用算法 $H2$ 求解，这一步的计算时间为 $f_{H2}(g_1(d(I_1)))$. 这样构造的算法对 A1 问题任意实例 I_1 的计算时间不超过 $g_1(d(I_1))+f_{H2}(g_1(d(I_1)))$，满足定义 2.4，A1 多项式归约为 A2 问题. 由此得出定义 2.4 比定义 2.3 更广泛.

用例 2.9 来理解定义 2.4.

例 2.9（续例 2.3） 适定性问题多项式归约为三精确覆盖问题.

例 2.3 定义的适定性问题为：$x_1,x_2,\cdots,x_n$ 为逻辑变量，$C_1,C_2,\cdots,C_m$ 为句子. 现在构造一个三精确覆盖问题，共分 3 个部分构造三精确覆盖的元素集合 S.

第 1 部分对所有的逻辑变量及其"非"进行复制，对应每一句子复制一次，共 $2mn$ 项，记成

$$U=\{x_i^j,\bar{x}_i^j \mid i=1,2,\cdots,n;j=1,2,\cdots,m\}.$$

第 2 部分包含 3 种类型的元素共 $2mn$ 项.

$$V=\{a_i^j \mid i=1,2,\cdots,n;j=1,2,\cdots,m\}\cup\{v_j \mid j=1,2,\cdots,m\}$$
$$\cup\{c_i^j \mid i=1,2,\cdots,n-1;j=1,2,\cdots,m\}.$$

第 3 部分同第 2 部分的构造基本相同，

$$W=\{b_i^j \mid i=1,2,\cdots,n;j=1,2,\cdots,m\}\cup\{w_j \mid j=1,2,\cdots,m\}$$
$$\cup\{d_i^j \mid i=1,2,\cdots,n-1;j=1,2,\cdots,m\}.$$

于是，S 中共包含 $6mn$ 个元素. 3 元素子集族 F 仅由下列 3 部分子集族组成：

第 1 部分：由 U,V,W 中各选一个元素，组成 (x_i^j,a_i^j,b_i^j)，$i=1,2,\cdots,n;j=1,2,\cdots,m$ 或 $(\bar{x}_i^j,a_i^{j+1},b_i^j)$；$i=1,2,\cdots,n;j=1,2,\cdots,m$，其中，定义 $a_i^{m+1}=a_i^1$. F 中包含 a 和 b 类型的元素仅有上面这些组合形式，则对任意 i，任何覆盖 $a_i^j,b_i^j,i=1,2,\cdots,n;j=1,2,\cdots,m$ 的所有 3 元素子集族中必是同由 a 和 b 类型的元素与 $x_i^j,j=1,2,\cdots,m$ 组合，即 (x_i^j,a_i^j,b_i^j)，$j=1,2,\cdots,m$，或是同由 a 和 b 类型的元素与 $\bar{x}_i^j,j=1,2,\cdots,m$ 组合，即 $(\bar{x}_i^j,a_i^{j+1},b_i^j)$，$j=1,2,\cdots,m$. 这一部分包含 $2mn$ 个元素.

第 2 部分：由 (y_j,v_j,w_j)，$j=1,2,\cdots,m$ 组成，其中，y_j 是句子 C_j 中的一个逻辑变量. 由于 U 包含所有逻辑变量的"是"及其"否"，自然包含 y_j. 因此，第 2 部分同第 1 部分仅在这一位置可能出现相同元素. 由 F 的第 1、第 2 部分组成，知 $y_j\in C_j\cap U$. 这一部分包含 m 个元素.

第 3 部分：(y_k,c_i^j,d_i^j)，$i=1,2,\cdots,n-1;j=1,2,\cdots,m,k=1,2,\cdots,m$，其中，$y_k\in U$. 这一部分包含 $(n-1)mm$ 个元素.

3 元子集族 F 中共有 $m^2(n-1)+m+2mn$ 个子集.

假设适定性问题的一个实例由 n 个逻辑变量的 m 个句子组成，实例的输入长度由 m 个句子中的逻辑变量值确定，因此实例的输入长度为 mn. 上面已构造三精确覆盖问题一个特殊情况，这是一种映射，简单估计三精确覆盖问题实例的输入长度不超过 $|S||F|$，其中 $|S|$

和$|F|$分别表示集合 S 和 F 中元素的个数,即不超过 $6mn(m+2mn+m^2n)$,定义 2.4 中的多项式函数选为 $g_1(x)=6x^3+18x^2$.

当 3 元素族 F 中存在 S 的一个三精确覆盖实例时,必包含第 2 部分的 m 组,取三精确覆盖中第 2 部分 y_j 的逻辑变量为"真"值,即,当 $y_j=x_i\in C_j$ 时,$x_i=1$;当 $y_j=\bar{x}_i\in C_j$ 时,$x_i=0$. 则对应适定性问题的一个"是"实例. 反之,对适定性问题的任何一个"是"实例,按上面的构造,对应三精确覆盖一个"是"实例,其三精确覆盖是:在 F 集合中的第 2 部分(y_j,v_j,w_j),$j=1,2,\cdots,m$ 中,取 y_j 为 C_j 中取值为"真"的逻辑变量;在 C_j 中逻辑变量 y_j 可能有两种形式 $y_j=x_i$ 或 $y_j=\bar{x}_i$. 以 $y_j=x_i=1$ 的情况来讨论(同法讨论 $y_j=\bar{x}_i=1$ 的情况):在 F 的第 1 部分中,取$\bar{x}_i^j\,(j=1,2,\cdots,m)$对应的所有 3 元组,至此,共选取$(n+1)m$ 个 3 元组;最后,在 F 的第 3 部分中,选取覆盖 S 中 U,V,W 的余下部分的$(n-1)m$ 个 3 元组. 由此,上面构造的问题可用三精确覆盖问题的一个算法求解. 因此,适定性问题可由多项式归约为三精确覆盖问题.

定义 2.5 (1) 对于判定问题 A,若 A∈NP 且 NP 中的任何一个问题可在多项式时间内归约为 A,则称 A 为 NP 完全问题(NP-complete 或 NPC). (2)对于判定问题 A,若 NP 中的任何一个问题可在多项式时间内归约为判定问题 A,则称 A 为 NP 困难问题(NP-hard 或 NPH).

所有 NP 完全问题的集合记为 NPC,即 A 为 NP 完全问题可以表示为 A∈NPC;所有 NP 困难问题的集合记为 NPH,即 A 为 NP 困难问题可以表示为 A∈NPH. 有时为了方便也直接说问题 A 是 NPC(或 NPH). NPC 和 NP-hard 两者的区别是:验证一个问题 A 是否为 NPC 的关键有两点,一是 NP 中的任何一个问题是否可在多项式时间内归约为 A,一是判断 A 是否属于 NP,即是否存在一个字符串(其输入长度为实例输入长度的多项式函数)和一个多项式时间的验证算法;验证一个问题 A 是否为 NP-hard 则无须判断 A 是否属于 NP. 根据定义 2.5 可知 NPC⊂NPH.

根据多项式转换,多项式归约,NPC,NPH 等的定义,当我们已知一个问题属于 NPC 或 NPH 时,如果再遇到一个新问题,只要已知问题可以多项式归约为新问题,则知新问题属于 NPH;当还可以验证新问题属于 NP 时,则该问题属于 NPC. 由此可知,研究一个新问题的复杂性,往往借助于已知问题的复杂性和多项式归约. 下面通过例子了解这一过程.

例 2.10 适定性问题是 NPC

证明可参阅文献[6]或文献[17]等. □

我们认为该问题是已知的. 下面是一种比较典型的研究方法,如果可能的话,将适定性问题归约到所研究的问题. 问题复杂性也就由此而得到.

例 2.11 三精确覆盖属于 NPC

由例 2.6 和例 2.9 得结论. □

例 2.12 子集和判定问题属于 NPC

子集和判定问题的解对应一个 n 维 0-1 分量的向量,于是,很易验证属于 NP.

例 2.8 已证明三精确覆盖多项式转换为子集和判定问题，所以，子集和判定问题属于 NPC. □

子集和判定问题的一个特殊情况，包的容积 $b=\frac{1}{2}\sum_{j=1}^{n}c_j$，此时，对应一类新的问题——集合划分(partition) 问题：

给定整数 $c_1,c_2,\cdots,c_n$，是否存在$\{1,2,\cdots,n\}$ 的一个子集 S，使得 $\sum_{j\in S}c_j=\sum_{j\notin S}c_j$？

例 2.13 集合划分问题是 NPC

因子集和判定问题属于 NP，所以，集合划分问题属于 NP. 例 2.12 已证明子集和判定问题属于 NPC，只须证明，子集和判定问题可以多项式转换为集合划分问题.

对给定的子集和判定问题的任何一个实例，$c_1,c_2,\cdots,c_n,b$，构造集合划分问题的实例 $c_1,c_2,\cdots,c_n,c_{n+1}=2M,c_{n+2}=3M-2b$，其中，$M=\sum_{j=1}^{n}c_j>b$.

显然，这个映射满足定义 2.3 的(1). 存在$\{1,2,\cdots,n\}$ 的子集 S，使得 $\sum_{j\in S}c_j=b$ 的充分必要条件是存在$\{1,2,\cdots,n+2\}$ 的一个子集 S' 使得 $\sum_{j\in S'}c_j=\sum_{j\notin S'}c_j$.

下面证明这个映射满足定义 2.3 的(2).

充分性：由于 $5M-2b>M=\sum_{j=1}^{n}c_j$，因此存在$\{1,2,\cdots,n\}$ 的一个子集 S，使得

$$\sum_{j\in S}c_j+c_{n+2}=\sum_{j\notin S,j\neq n+1,n+2}c_j+c_{n+1},\tag{2.5}$$

所以，计算上式可得 $\sum_{j\in S}c_j=b$.

必要性：由式(2.5)，令 $S'=S\cup\{n+2\}$，得结论. □

例 1.10 装箱问题的判定问题可以叙述为：给定 K 及 n 个物品，尺寸为 $0<b_1,b_2,\cdots,b_n\leqslant 1$，是否能在 K 只尺寸为 1 的箱子中装入这 n 个物品？

例 2.14 装箱判定问题是 NPC

装箱判定问题属于 NP 易证. 我们通过集合划分问题多项式转换为装箱判定问题而证明结论. 由例 2.13 的证明，当集合划分问题的所有整数为正时，问题还是 NPC. 对集合划分问题的任一实例：n 个正整数 $c_1,c_2,\cdots,c_n$，映射装箱问题的一个判定实例：n 个物品的尺寸满足

$$b_i=2c_i\Big/\Big(\sum_{j=1}^{n}c_j\Big),\quad i=1,2,\cdots,n,$$

箱子尺寸 1，是否能在两个箱子内装完所有物品？显然上面的映射满足定义 2.3 的(1). 现验证定义 2.3 的(2) 满足. 证明下列推断：存在$\{1,2,\cdots,n\}$ 的一个子集 S 使 $\sum_{j\in S}c_j=\sum_{j\notin S}c_j$ 的充分必要条件是可在两个尺寸为 1 的箱子里装入尺寸为

$$b_i=2c_i\Big/\Big(\sum_{j=1}^{n}c_j\Big),\quad i=1,2,\cdots,n$$

的 n 个物品(不妨设 $b_i \leqslant 1$).

充分性：因为 $\sum_{j=1}^{n} b_j = 2$,则每个箱子都无空隙.记第 1 个箱子中的物品集为 S,有 $\sum_{j\in S} c_j = \sum_{j\notin S} c_j = 1$.

必要性:明显. □

例 1.11 的约束单机排序问题对应的判定问题是:给定 $d_i, i=1,2,\cdots,n, c_t, t=1,2,\cdots$ 和 T,在模型的约束条件下,能否在 T 个工作日内完成 n 个产品的加工?

例 2.15　约束单机排序的判定问题是 NPC

约束单机排序的判定问题属于 NP 是很容易验证的. 当 $c_t(t=1,2,\cdots)$全部相同时,便成为装箱问题的一种特殊情况,因此,装箱问题就可多项式转换为约束单机排序问题的这一特殊情况. 所以,结论成立. □

本章所介绍的 4 类问题：P,NP,NPC,NP-hard 的复杂性关系可用示意图 2.1 表示.

示意图 2.1 表示 P⊆NP,NPC⊆NP-hard,NP 与 NP-hard 的公共部分为 NPC. 在 NP 中,除 P 和 NPC,还有一部分问题的复杂性是未知的. 由于 NPC 里包含很多著名的组合优化问题,经过几代数学家的努力,迄今没有找到多项式时间算法,人们猜想：NPC 中的任何一个问题都没有多项式时间算法,即 P∩NPC=∅. 虽说到目前为止,这个猜想既没有被证明，也没有被否定，但大家一般都认为它应当是成立的，所以大家一般都接受这个假设!

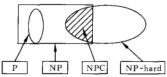


图 2.1　复杂性示意图

假设　P≠NP .

按 NPC 的定义容易证明:若 P∩NPC≠∅,则 NPC=P. 此外,我们还可以定义所谓的强 NPC(strongly NPC)问题,从而定义强 NPC 问题类等. 所谓强 NPC 问题，是指如果该 NPC 问题存在一个伪多项式时间算法(其定义见 1.4 节)，则必有 P=NP. 也就是说，在承认 P≠NP 的假设下，强 NPC 问题也就是 NPC 问题中那些不存在伪多项式时间算法的问题. 可见，强 NPC 问题差不多也就是 NPC 中最难的那些问题. 可以证明，前面介绍的 TSP 问题、适定性问题等都是强 NPC 问题，而背包问题则不是强 NPC 问题. 另外,对于组合优化问题,若其判定问题是 NPC 或 NP-hard,则知优化问题的难度不低于判定问题,此时,称组合优化问题为 NP-hard. 有关算法及复杂性的理论的详细讨论,可以参考文献[6].

2.2　算法设计与分析

通过复杂性理论,我们可以判断一些问题是属于 P 还是 NP-hard,进而采用相应的处理方法. 对于 P 问题,我们希望设计计算复杂性尽量低的算法. 对于 NP-hard 问题,不能期望在多项式时间内求得其任意实例的最优解,那么一个自然的想法是通过近似算法

(approximation algorithm)在多项式时间内得到问题的一个近似解. 近似算法的优劣一般通过其"最坏情形性能比"来衡量.

定义 2.6 对于使目标函数 f 最大的优化问题,记 I 是这个优化问题的一个实例,H 是一个多项式时间算法. $f(I)$是对于实例 I 的最优目标函数值,$f_H(I)$是应用算法 H 所得到的目标值. 如果存在一个实数 $\rho\leqslant 1$,使得对于任意实例 I,都有 $f_H(I)\geqslant\rho f(I)$,那么称算法 H 是 ρ-近似算法,ρ 是算法 H 的一个近似比. 如果 ρ 是所有可能值中最大的一个,则称 ρ 是算法 H 的最坏情形性能比,或简称为最坏比. 类似地,对于使目标函数最小的优化问题,我们也可以给出相应的定义.

本章将介绍算法设计与分析中一些常用的方法,它们不仅适用于求 P 问题,同时也广泛应用于 NP-hard 问题的近似算法设计中.

2.2.1 贪婪算法

很难对贪婪算法给出一个严格的定义. 一般说来,贪婪算法逐步地得到问题的解,在每一步决策中,只考虑如何在当前状态下优化某个目标,并且每步的决策作出后不再更改. 贪婪算法往往所需要的步骤不多,而且每步的决策也比较直观、简单. 对于同一个问题,从不同的角度,可以设计不同的贪婪算法,有些可以得到问题的准确解,也可能只得到问题的近似解. 在下一章中,我们可以看到贪婪算法可以在多项式时间内求解最小支撑树问题. 这里先给一个简单的例子.

例 2.16(续例 1.7) 背包问题的贪婪算法

首先背包问题属于 NPC,因此不能期望得到一个多项式时间算法,我们考虑利用贪婪算法对其设计近似算法.

一个自然的想法是把所有物品按照收益体积比 $e_j=c_j/a_j$ 降序排列,然后把物品依次放入背包之中,直至背包所剩容积小于当前物品的体积. 但是这个贪婪算法的性能很差,其近似比可以任意接近于 0. 比如这样的例子:背包的容积为 1,有 2 个工件,它们的收益和体积分别为 $c_1=\varepsilon,a_1=1/(2\varepsilon),c_2=1,a_2=1$,这里 $0<\varepsilon<1$. 贪婪算法会装入第一个物品,最优解是装入第二个物品,那么贪婪算法的最坏比不好于 $\varepsilon/1$,当 $\varepsilon\to 0$ 时,最坏比趋近于 0. 这个算法的时间复杂度为 $O(n\log n)$. □

我们可以把这个贪婪算法做适当的改进:把贪婪算法得到的解和单一物品的最大收益进行比较,选取较优的一个. 这个算法的时间复杂度也是 $O(n\log n)$. 我们有下面的定理.

定理 2.1 改进的贪婪算法对于背包问题的最坏比是 $\dfrac{1}{2}$.

证明 设背包问题的最优值为 z^*,改进的贪婪算法得到的目标值为 z. 容易看出,z^* 不大于贪婪算法的目标值与排在它后面物品的收益之和. 而这两部分都不大于 z. 因此 $z/z^*\geqslant 1/2$.

再考察下面的例子. 背包容积 $b=2M(M>2)$,3 个物品,其中第 1 个物品对应参数为

$a_1=1, c_1=2$,后两个物品相同,$a_2=a_3=c_2=c_3=M$. 那么改进贪婪算法装入前 2 个物品,$z=2+M$. 显然最优解是装入后 2 个物品,$z^*=2M$. 当 $M\to\infty, z/z^*\to 2$. □

贪婪算法可以看作是一种最"自然"的算法,它可以处理一些比较简单特殊的问题,同时,一些复杂的算法往往也基于一个或多个贪婪算法. 下面介绍两种设计精巧,功能强大的的算法设计技术:动态规划(dynamic programming)方法和线性规划(linear programming)方法.

2.2.2 动态规划

动态规划将决策问题的全过程恰当地划分为若干个相互联系的子过程(每个子过程为一个阶段),以便按照一定的次序去求解. 阶段一般是根据时间和空间的自然特征来划分,以便于问题的求解为目的. 在决策问题中,状态(state)表示每个阶段开始时所处的自然状况或客观条件,描述过程状态的变量称为状态变量. 当过程处于某个阶段的某个状态时,从该状态演变为下一个阶段某状态的选择,称为决策(decision). 描述决策的变量称为决策变量. 相邻两个状态下的状态变量和决策变量的关系称为状态转移方程. 益损值一般用准则函数(criterion function)来度量,即准则函数是衡量策略好坏的尺度,有些书上也称为指标函数. 我们所要介绍的动态规划中,决策问题都可以划分为无后效性的多个阶段,即当某阶段的状态一旦确定,则此后过程的演变不再受此前各状态和决策的影响,或者说"未来与过去无关".

最优化原理是 Bellman 于 20 世纪 50 年代提出的,其表述如下:"全过程的最优策略具有这样的性质:不管该最优策略上某状态以前的状态和决策如何,对该状态而言,余下的诸决策必定构成最优子策略."即:最优策略的任一后部子策略都是最优的. 下面我们介绍应用动态规划的两个例子.

例 2.17 资源分配问题

某公司现有 M 台设备准备分配给该公司所属的 N 家工厂. 当分配 x_k 台设备给工厂 k 时,工厂 k 利用这些设备为公司创造的利润为 $g_k(x_k)(1\leqslant k\leqslant N)$. 假设所有 $g_k(x_k)\geqslant 0$,应当如何分配设备资源,使得公司总利润最大?

该问题可以用整数规划描述如下:

$$\max\ z=\sum_{k=1}^{N} g_k(x_k)$$

$$\text{s.t.}\ \sum_{k=1}^{N} x_k = M$$

$$x_k\in\mathbb{Z}^+$$

由于 $g_k(x_k)$不一定是线性函数,因此上述问题可能是非线性整数规划. 我们通过如下过程把它转化为多阶段决策(multistage decision making)过程,并建立动态规划模型.

共有 N 个工厂,可以把问题分解为 N 个阶段:当阶段 $k=N$ 时,把手中设备分配给工厂

N；当阶段 $k=N-1$ 时，把手中设备分配给工厂 $N-1$；依此类推，在任意阶段 k 时，把手中设备分配给工厂 k；当阶段 $k=1$ 时，把手中设备分配给工厂 1.

状态变量 u_k 可以如下选取：u_k 表示第 k 阶段初分配者手中拥有的设备台数. 由题意可知 $u_1=M, u_{N+1}=0$.

决策变量 x_k 可以如下选取：x_k 表示第 k 阶段分配给工厂 k 的设备台数（$0\leqslant x_k\leqslant u_k$）.

因此，状态转移方程为 $u_{k+1}=u_k-x_k$. 阶段 k 的准则函数为 $g_k(x_k)$.

用 $f_k(u_k)$ 表示将手中现有资源 u_k 分配给工厂 $k, k+1, \cdots, N$ 时的最大利润，则有如下动态规划基本方程：

$$f_k(u_k)=\max_{0\leqslant x_k\leqslant u_k}\{g_k(x_k)+f_{k+1}(u_{k+1})\},\quad k=N,N-1,\cdots,1,$$

初始条件为 $f_{N+1}(u_{N+1})=0$. 此时原问题即为计算 $f_1(M)$.

当给定具体数据时，就可以通过递推求解上述方程得到问题的最优解. 例如，$M=4$，$N=3$，且 $g_k(x_k)$ 通过下表给出.

表 2.4　资源分配利润表

设备数 x_k \ $g_k(x_k)$ \ 工厂 k	1	2	3
0	0	0	0
1	4	2	3
2	6	5	5
3	7	6	7
4	7	8	8

此时 $u_1=4, u_4=0, 0\leqslant u_k\leqslant 4(k=2,3)$. 递推求解如下：

边界条件 $f_4(u_4)=f_4(0)=0$.

$k=3$ 时：因为 $g_k(x_k)$ 是增函数，所以

$$f_3(u_3)=\max_{0\leqslant x_3\leqslant u_3}\{g_3(x_3)+f_4(0)\}=g_3(u_3)$$

$$f_3(0)=g_3(0)=0;\quad f_3(1)=g_3(1)=3;\quad f_3(2)=g_3(2)=5;$$

$$f_3(3)=g_3(3)=7;\quad f_3(4)=g_3(4)=8.$$

$k=2$ 时：

$$\begin{aligned}f_2(u_2)&=\max_{0\leqslant x_2\leqslant u_2}\{g_2(x_2)+f_3(u_3)\}\\&=\max_{0\leqslant x_2\leqslant u_2}\{g_2(x_2)+f_3(u_2-x_2)\}.\end{aligned}$$

因此

$$\begin{aligned}f_2(0)&=\max_{0\leqslant x_2\leqslant 0}\{g_2(x_2)+f_3(0-x_2)\}\\&=g_2(0)+f_3(0)=0+0=0,\end{aligned}$$

$$f_2(1) = \max_{0 \leqslant x_2 \leqslant 1} \{g_2(x_2) + f_3(1 - x_2)\}$$
$$= \max\{g_2(0) + f_3(1), g_2(1) + f_3(0)\}$$
$$= \max\{0 + 3, 2 + 0\} = 3,$$

类似地,可以得到 $f_2(2)=5, f_2(3)=8, f_2(4)=10$.

$k=1$ 时:

$$f_1(u_1) = \max_{0 \leqslant x_1 \leqslant u_1} \{g_1(x_1) + f_2(u_2)\}$$
$$= \max_{0 \leqslant x_1 \leqslant u_1} \{g_1(x_1) + f_2(u_1 - x_1)\}.$$
$$f_1(4) = \max_{0 \leqslant x_1 \leqslant 4} \{g_1(x_1) + f_2(4 - x_1)\}$$
$$= \max\{g_1(0) + f_2(4), g_1(1) + f_2(3), g_1(2) + f_2(2), g_1(3) + f_2(1), g_1(4) + f_2(0)\}$$
$$= \max\{0 + 10, 4 + 8, 6 + 5, 7 + 3, 7 + 0\} = 12.$$

由上面计算可知:$f_1(4)=12$ 为最大利润.进一步分析计算过程,可知 $f_1(4)=12$ 在$x_1=1$, $x_2=2, x_3=1$ 时成立.因此我们得到最优解 $x_1^*=1, x_2^*=2, x_3^*=1$,最大利润为 12. □

例 2.18(续例 2.16) 背包问题的动态规划算法

背包问题的数学模型表示为

$$\max \sum_{i=1}^{n} c_i x_i$$
$$\text{s. t.} \sum_{i=1}^{n} a_i x_i \leqslant b,$$
$$x_i \in \{0,1\}, \quad i = 1,2,\cdots,n,$$

假设所有的参数都是整数.

共有 n 个物品,可以把问题分解为 n 个阶段,在第 k 个阶段,决定是否把第 k 个物品放入背包.

状态变量 u_k 可以如下选取:u_k 表示第 k 阶段初背包剩余的容积.可知 $u_1=b, 0\leqslant u_k \leqslant b, k=1,2,\cdots,n$.

决策变量 x_k 可以如下选取:$x_k=1$ 表示把第 k 个物品放入背包;否则,$x_k=0$.因此,状态转移方程为 $u_{k+1}=u_k - x_k a_k$.

用 $f_k(u_k)$表示背包剩余容积为 u_k 时,选取物品 $k, k+1, \cdots, n$ 所能得到的最大收益,则有如下动态规划基本方程:

$$f_k(u_k) = \begin{cases} f_{k+1}(u_k), & \text{若 } a_k > u_k, \\ \max\{f_{k+1}(u_k), f_{k+1}(u_k - a_k) + c_k\}, & \text{若 } a_k \leqslant u_k. \end{cases}$$

初始条件为 $f_{n+1}(x_{n+1})=0$.此时原问题即为计算 $f_1(b)$.

注意到对于任意一个阶段,状态变量的个数不超过 b,因此每一阶段的操作次数为

$O(b)$，共有 n 个阶段，因此总的运算时间为 $O(nb)$，这是一个伪多项式时间算法. □

动态规划方法本质上是一种隐式枚举方法，它聪明地省略了一些中间过程不必要的运算，所以很好地解决了一些优化问题. 但当问题规模较大时，也会遇到维数障碍（维数灾难）的问题，需要具体问题具体分析.

2.2.3 线性规划方法——全幺模矩阵

许多组合优化问题，特别是网络优化问题，可以用整数线性规划(ILP)模型来表示：

$$\text{(ILP)}\quad \begin{aligned} \min\ & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.}\ & \boldsymbol{Ax}=\boldsymbol{b}, \\ & \boldsymbol{x}\geqslant \boldsymbol{0},\ \boldsymbol{x}\in\mathbb{Z}^n. \end{aligned} \tag{2.6}$$

其中，$\boldsymbol{c}$ 为 n 维列向量，$\boldsymbol{A}$ 为 $m\times n$ 矩阵，$\boldsymbol{b}$ 为 m 维列向量，$\boldsymbol{x}$ 为 n 维决策变量，$\mathbb{Z}$ 表示整数集合. 模型中的系数 $\boldsymbol{A}$，$\boldsymbol{b}$ 和 $\boldsymbol{c}$ 都是整数. 如果去掉 $\boldsymbol{x}\in\mathbb{Z}^n$ 的约束，我们得到一个线性规划模型：

$$\text{(LP)}\quad \begin{aligned} \min\ & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{s.t.}\ & \boldsymbol{Ax}=\boldsymbol{b}, \\ & \boldsymbol{x}\geqslant \boldsymbol{0}, \end{aligned} \tag{2.7}$$

线性规划(2.7)称为整数线性规划(2.6)对应的线性规划松弛问题.

我们知道，一般的整数规划问题是很难求解的. 与之相对照，线性规划问题却可以很好地求解. 一个非常自然的问题是：是否可以借助其线性规划松弛问题来求解或近似求解整数线性规划问题？可以考虑的方法有很多，这里要介绍一类特殊的整数线性规划问题，它和其线性规划松弛问题有相同的最优解！这样，我们就可以通过求解线性规划松弛问题得到原问题的最优解.

线性规划松弛问题的最优解什么时候一定是整数解呢？也就是说，在整数规划中，当其约束矩阵 $\boldsymbol{A}$ 满足什么样的特殊性质时，其线性规划松弛问题与原问题有相同的最优解呢？我们先给出全幺模矩阵的定义.

定义 2.7 如果一个矩阵 $\boldsymbol{A}$ 的任何子方阵的行列式的值都等于 0，1 或 -1，则称 $\boldsymbol{A}$ 是全幺模的(totally unimodular，又译为全单位模的)，或称 $\boldsymbol{A}$ 是全幺模矩阵.

由全幺模矩阵的定义，不难得到下面的性质.

定理 2.2 下列命题等价：

(1) $\boldsymbol{A}$ 是全幺模矩阵.

(2) $-\boldsymbol{A}$ 是全幺模矩阵.

(3) $\boldsymbol{A}^{\mathrm{T}}$ 是全幺模矩阵.

(4) $(\boldsymbol{A},\boldsymbol{A})$ 是全幺模矩阵.

(5) $(\boldsymbol{A},\boldsymbol{I})$ 是全幺模矩阵. □

于是，上面的定理有如下推论：如果在式(2.7)所示的线性规划中 $\boldsymbol{A}$ 为全幺模矩阵，且该线性规划问题有最优解，则一定存在整数最优解. 当采用单纯形算法求解时，每次旋转变

换实际上是从一个极点变换到另一个极点，因此求解过程中和最后得到的解都是整数解. 也就是说，可以认为 $\boldsymbol{A}$ 为全幺模矩阵时的整数规划问题实际上等价于对应的线性规划松弛问题.

显然，全幺模矩阵的元素只能取 0,1 和 −1. 下面我们介绍一个判定矩阵是否满足全幺模性质的充分条件.

定理 2.3 设 $\boldsymbol{A}$ 是由 0,1 和 −1 组成的矩阵，如果下面两个条件同时成立，则 $\boldsymbol{A}$ 为全幺模矩阵.

(1) $\boldsymbol{A}$ 的每一列至多含有两个非零元素.

(2) $\boldsymbol{A}$ 的行可以分为 $A1$, $A2$ 两个集合，使得：如果 $\boldsymbol{A}$ 的一列中有两个符号不同的元素，则相应的两行在同一集合 $A1$ 或 $A2$ 中；如果 $\boldsymbol{A}$ 的一列中有两个符号相同的元素，则相应的两行分别位于两个集合 $A1$ 和 $A2$ 中.

证明 如果矩阵 $\boldsymbol{A}$ 满足条件(1)和(2)，则 $\boldsymbol{A}$ 的任意子矩阵仍然满足条件(1)和(2). 所以，只需证明当 $\boldsymbol{A}$ 为方阵且满足条件(1)和(2)时，$\det(\boldsymbol{A})=0,1$ 或 -1 即可. 下面用数学归纳法证明这一点.

当 $\boldsymbol{A}$ 为 1 阶方阵时，显然 $\det(\boldsymbol{A})=0,1$ 或 -1. 假设结论对任意 $n-1$ 阶方阵均成立，下面考虑 $\boldsymbol{A}$ 为 n 阶方阵的情形. 有且只有以下 3 种可能：

假设 $\boldsymbol{A}$ 的某一列元素全为 0，则 $\det(\boldsymbol{A})=0$.

假设 $\boldsymbol{A}$ 的某一列元素只有一个不为 0，则其行列式的值可以按该元素展开进行计算，即 $\det(\boldsymbol{A})=\det(\boldsymbol{B})$ 或 $-\det(\boldsymbol{B})$，其中 $\boldsymbol{B}$ 为该非零元的余子式. 由于 $\boldsymbol{B}$ 为 $n-1$ 阶方阵，由归纳假设有 $\det(\boldsymbol{B})=0,1$ 或 -1，所以 $\det(\boldsymbol{A})=0,1$ 或 -1.

假设 $\boldsymbol{A}$ 的每一列均含有两个非零元素，则由条件(2)可知

$$\sum_{i\in A1} a_{ij} = \sum_{i\in A2} a_{ij}, \quad \forall j = 1,2,\cdots,n.$$

也就是说，$\boldsymbol{A}$ 的一些行的线性组合等于另一些行的线性组合. 由行列式的性质易知 $\det(\boldsymbol{A})=0$.

综上，定理得证. □

推论 (1) 一个有向图的关联矩阵为全幺模矩阵.

(2) 一个无向图的关联矩阵为全幺模矩阵，当且仅当该图为二部图.

当采用整数规划来描述网络优化问题时，其约束矩阵一般是有向图的关联矩阵或它的简单变形，一般都是全幺模矩阵. 因此可以认为，许多网络优化问题都是一类特殊的整数规划，其线性规划松弛与原问题有相同的最优解.

最后，我们不加证明地介绍一个判定全幺模矩阵的充要条件. 它是通过 Euler 矩阵来描述的，由 Camion 于 1965 年给出.

定义 2.8 如果一个矩阵的每行元素和、每列元素和都是偶数，则称该矩阵为 Euler 矩阵.

定理 2.4 设 $\boldsymbol{A}$ 是由 0,1 和 −1 组成的矩阵,则 $\boldsymbol{A}$ 为全幺模矩阵的充要条件是 $\boldsymbol{A}$ 的每一个 Euler 子方阵的所有元素的和为 4 的倍数. □

2.2.4 两分法

在后面章节的网络优化算法中,两分法是设计多项式算法的一种常用方法.下面我们用一个一般化的问题来说明这种方法.

在讨论问题复杂性时,我们一直用判定问题去讨论,大量的组合最优化问题都有目标函数,这一点不同于判定问题.上面的讨论都是将组合优化问题化成相应的判定问题,它们之间有什么关系呢?考虑定义 1.3 所定义的组合优化问题:

$$\begin{aligned}&\min\ f(\boldsymbol{x})\\&\text{s.t.}\ \boldsymbol{g}(\boldsymbol{x})\geqslant \boldsymbol{0},\\&\quad \boldsymbol{x}\in D.\end{aligned}\tag{2.8}$$

其中,$f(\boldsymbol{x})$为目标函数,$\boldsymbol{g}(\boldsymbol{x})$为约束函数,$\boldsymbol{x}$ 为决策变量,D 表示有限个点组成的集合.其对应的判定问题为:给定 z,是否有

$$\{\boldsymbol{x}\mid f(\boldsymbol{x})\leqslant z,\boldsymbol{g}(\boldsymbol{x})\geqslant \boldsymbol{0},\boldsymbol{x}\in D\}\neq\varnothing?\tag{2.9}$$

定理 2.5 如果对于给定的任意实例 I,优化问题(2.8)的目标值为整数,并且有上、下界 L 和 U,L 和 U 满足 $\max\{|L|,|U|\}=O(a^{d(I)})$,这里 a 为常数,$d(I)$为实例 I 的输入长度.那么求优化问题(2.8)的最优值多项式时间可解的充分必要条件为判定问题(2.9)多项式时间可解.

证明 不妨设 $L=0$,否则可以令目标函数为 $f(x)-L$,从而得到一个等价的优化问题.

必要性:当优化问题(2.8)存在一个多项式时间算法得到其最优值,那么调用这个算法,然后比较所得到的最优值和给定的 z 的大小关系,从而回答判定问题(2.9),这个过程可以在多项式时间内完成.

充分性:当判定问题(2.9)存在一个多项式时间算法,那么下面的两分搜索算法可以多项式时间内求得优化问题(2.8)的最优值.

二分算法

STEP1 令 $z_l=0,z_u=U;K=\left\lceil\frac{z_l+z_u}{2}\right\rceil$.

STEP2 若 $z_u-z_l\leqslant1$ 时停止,进入 STEP3.否则若 $z=K$ 时,判定问题(2.9)为"是"实例,则令 $z_l=z_l,z_u=K$;若判定问题(2.9)为"否"实例,则令 $z_l=K,z_u=z_u$;$K=\left\lceil\frac{z_l+z_u}{2}\right\rceil$,重复 STEP2.

STEP3 若 $z=z_l$ 时,判定问题(2.9)为"是"实例,则最优值为 z_l;否则,最优值为 z_u.

二分算法的迭代次数不超过 $\log U+1=O(\log a^{d(I)})=O(d(I)\log a)$. 每次迭代调用一次求解判定问题(2.9)的多项式时间算法，所以总的运算时间仍然是多项式的. □

2.2.5 网络搜索算法

搜索算法是网络算法的基础，它们的目的是搜索网络中满足某些性质的节点的集合，这里介绍一个常用的网络搜索算法.

例 2.19 给定一个无圈有向网络 $G=(V,A)$，$|V|=n$，$|A|=m$ 对其节点进行编号，使得对于图中任意弧 $(i,j)\in A$ 有节点 i 的编号小于节点 j 的标号，这称为图的拓扑排序.

由于网络中没有有向圈，因此至少有一个节点的入度为 0，对其标号为 1，删去这个节点及与其关联的弧. 可以看出残余的网络仍然无圈，至少有一个节点的入度为 0，对其标号为 2，删去这个节点及与其关联的弧. 依次进行，直至对所有节点进行标号. 在每次标号中，我们只对残余网络中没有入弧的一个节点进行标号，那么对于任意弧 $(i,j)\in A$，节点 i 一定在节点 j 之前被标号并删除，因此节点 i 的编号小于节点 j 的标号，因此这是一个拓扑排序. 对于一般有向网络来说，如果对其进行拓扑排序，在某一步骤中所有节点的入度都不为 0，那么这个图中一定包含有向圈，因此拓扑排序算法也用来判断网络中是否含有有向圈.

用 $indegree(i)$ 表示节点 i 当前的入度，用 $List$ 来存放当前入度为 0 的节点的集合. 拓扑排序算法具体算法描述如下：

拓扑排序算法

STEP0 对任意节点 i，计算其入度 $indegree(i)$，令 $List=\varnothing$，$label=1$.

STEP1 如果 $indegree(i)=0$，令 $List=List\cup\{i\}$，如果 $List=\varnothing$，转入 STEP2；否则，从 $List$ 中选出一个节点 i，对其标号为 $label$，把它从 $List$ 中删除，令 $label=label+1$. 如果 $(i,j)\in A$，令 $indegree(j)=indegree(j)-1$，重复 STEP1.

STEP2 如果 $label\leqslant n$，那么网络中有有向圈；否则，网络为无圈图，节点的标号为其拓扑排序.

STEP0 需要对所有的节点和弧扫描一遍，该步的计算复杂度为 $O(m+n)$. STEP1 最多包含 n 次循环，每次循环包含两个过程：寻找入度为 0 的节点 i 和修改满足 $(i,j)\in A$ 的节点 j 的入度，如果节点 j 的入度为 0，那么把它加入 $List$ 中. 对于给定的 $List$，寻找入度为 0 的节点可以在 $O(1)$ 时间完成. 第二个过程需要进行 $O(d^+(i))$（节点 i 的出度）次操作. STEP1 的计算复杂度为 $O\left(n+\sum_{i=1}^{n}d^+(i)\right)=O(m+n)$，因此总的复杂度为 $O(m+n)$. □

2.3 小 结

计算复杂性理论是组合最优化的基础.只有了解所研究问题的复杂性才可能有针对性地设计算法,提高工作效率,起到事半功倍的作用.有些实际应用人员往往忽略这一基础工作,这不仅仅是因为数学的难度,更重要的是一种模糊观念在起作用.这种观念是:无论解的效果如何,能找一个算法或比较已有的各种方法求得一个解即可.这在组合最优化的研究中是不可取的.

在算法设计与分析的方面,本章讨论了一些常用的方法与技术,更多内容可以参阅文献[7,10].对于NPC,NPH问题研究的文献中,除了近似算法,启发式算法也是一种常见的研究方法.启发式算法一般指受到生物界、自然界的一些规律和现象启发而得到的算法,虽然一般而言,很难在理论上保证它们的性能,但对很多困难的优化问题,它们具有优异的实际效果.《现代优化计算方法》(参阅文献[18])是专门讨论近年来兴起的一些启发式算法的,有兴趣的读者可以参阅该书.

应该注意的是有些问题是多项式时间可解,而此时再采用近似算法或启发式方法势必造成解的效果较差或计算时间的浪费.一个多项式时间可解的问题一般不必用近似或启发式算法.本书后续章节将介绍的主要内容是网络优化问题的一些常用算法,主要是多项式时间算法,但也包括一些非多项式时间算法.我们希望读者不仅要理解算法,而且能进行算法的评价(主要考虑最坏情况分析,但偶尔我们也提及一些计算测试的经验性结果分析).

练 习 题

1. 证明:若存在计算TSP判定问题的多项式时间算法,则存在求TSP最优路径的多项式时间算法.

2. 证明集合覆盖问题属于NPC.集合覆盖问题为:给定一个 $m\times n$ 的0-1矩阵 $\boldsymbol{A}$,整系数 $\{c_1,c_2,\cdots,c_n\}$ 和一个整数 K,是否存在取值为0-1的变量 $(x_1,x_2,\cdots,x_n)^{\mathrm{T}}$ 使得 $\boldsymbol{Ax}\geqslant\boldsymbol{e}$ 且 $\sum_{j=1}^{n}c_jx_j\leqslant K$?(其中 $\boldsymbol{e}=(1,1,\cdots,1)^{\mathrm{T}}$).

3. 哈密顿圈(Hamilton circle)问题为:给定一个无向图 $G=(N,E)$,其中 $N=\{1,2,\cdots,n\}$ 为所有的节点组成的集合,$E=\{(i,j)\mid i,j\in N\}$ 为边集合,是否存在一个圈通过所有节点正好一次?假设哈密顿问题是NPC,证明:TSP属于NP-hard问题.

4. 证明下面问题是NP-hard问题:

$$\max\ \sum_{i=1}^{n}\sum_{j=1}^{n}c_{ij}x_{ij}$$

$$\text{s.t.} \sum_{i=1}^{n} x_{ij} = 1, \forall j,$$
$$\sum_{j=1}^{n} x_{ij} = 1, \forall i,$$
$$\sum_{i=1}^{n}\sum_{j=1}^{n} t_{ij} x_{ij} \leqslant T,$$
$$x_{ij} \in \{0,1\}, \forall i,j.$$

5. 假设运输问题中要求运输量都是整数，用整数规划描述运输问题，并证明其线性规划松弛问题存在整数最优解.

6. 用动态规划方法求解下列整数规划问题：

(1) $\min z = x_1^2 + x_2^2 + x_3^2 + x_4^2$

$\text{s.t. } x_1 + x_2 + x_3 + x_4 \geqslant 10,$

$x_1, x_2, x_3, x_4 \in \mathbb{Z}^+.$

(2) $\max z = 5x_1 + 10x_2 + 3x_3 + 6x_4$

$\text{s.t. } x_1 + 4x_2 + 5x_3 + 10x_4 \leqslant 10,$

$x_1, x_2, x_3, x_4 \in \mathbb{Z}^+.$

7. 考虑一个总期限为 $N+1$ 年的设备更新问题. 已知一台新设备的价值为 C 元，其 T 年末的残值为

$$S(T) = \begin{cases} N-T, & N \geqslant T, \\ 0, & N < T. \end{cases}$$

对有 T 年役龄的该设备，其年创收益为

$$P(T) = \begin{cases} N^2 - T^2, & N \geqslant T, \\ 0, & N < T. \end{cases}$$

试给出最优的设备更新方案：(1)建立动态规划模型；(2)当 $N=3, C=10$ 时，具体求解.

8. 用动态规划方法求解如下问题：假设有一台机器，它可能处于正常运行或损坏故障两种状态之一. 如果正常运行一周，可以得到利润 10 000 元；如果一周中出现损坏故障，本周利润为 0. 如果一周开始时机器处于正常运行状态，并采取预防性保护措施，则一周中出现损坏故障的概率为 0.4；如果不采取预防性保护措施，则一周中出现损坏故障的概率为 0.7. 然而，采取预防性保护措施需要花费 2000 元. 如果一周开始时机器处于损坏故障状态，可以有两种方案提供选择：第 1 是花费 4000 元修理费去修理，在这种情况下机器再次出现损坏故障的概率为 0.4；第 2 是花费 15 000 元购买一台新机器代替，在这种情况下机器可以保证在第 1 周正常运行. 假定第 1 周开始时为一台新机器，请你设计一个修理，采购，维护方案，使得前 4 周的总利润最大. 如果要求使前 5 周的总利润最大，方案又是如何呢？

9. 给定一个 n 阶无向图，设计一个算法判断该图是否为二分图并分析其复杂度.

10. 判定问题"一个无向图中是否有从一个起点出发经过其他节点以后再回到这个起点的一个圈?"属于P. 给出一个多项式时间的算法并估计算法的复杂性.

11. 平行机排序问题(parallel machine scheduling): n 个工件要在 m 台相同的机器上加工,工件 i 需要加工时间为 p_i, $i=1,2,\cdots,n$. 一台机器同时最多加工一个工件,一个工件同时最多只能在一台机器上加工,如何安排工件的加工顺序,使得最后完工时间最小? 证明这个问题属于NPC,并给出一个近似算法.

第 3 章 最小树与最小树形图

在例 1.1 中,我们介绍了公路连接问题:某一地区有 n 个主要城市,现准备修建高速公路把这些城市连接起来,使得从其中任何一个城市都可以经高速公路直接或间接到达另一个城市.假定已经知道了任意两个城市 i 和 j 之间修建高速公路的成本(费用)w_{ij}(>0),那么应如何决定在哪些城市间修建高速公路,使得总成本最小?

根据 1.2 节介绍的图与网络的概念,上述问题可以用网络图来建模.考虑以这 n 个城市为节点构成的完全图G(不妨假设为无向图),并令城市 i 和 j 之间弧上的权为 w_{ij}(当 (i,j) 之间不可能修建高速公路时,可以认为对应的权为无穷大).显然,准备修建的高速公路网正好构成这个完全图 G 的一个子图 T.为了使得从其中任何一个城市都可以经高速公路直接或间接地到达另一个城市,该子图 T 必须是连通的且包含所有 n 个城市对应的节点,即该子图 T 是 G 的连通的支撑子图.同时,为了使得总建设成本最小,该子图必须是无圈的,因为有圈时可以减少圈上的一条弧(即不修建对应的高速公路,减少建设成本),仍然可以保证通过该高速公路网从任意城市 i 到达城市 j.

也就是说,例 1.1 中的公路连接问题实际上就是在图 G 中求一个无圈连通支撑子图 T,使得 T 所包含的弧上的权的和(总成本)最小.在图论的术语中,我们把无圈连通图称为树,因此把上面这样一类问题称为最小树问题.

本章我们首先在无向网络的情形下介绍树的基本概念和性质,并介绍最小树问题的几个重要算法.然后,我们在有向网络的情形下讨论与最小树问题类似的最小树形图问题和最大分枝问题,并介绍相应的算法.

3.1 树的基本概念

树是一类既简单而又非常重要的图形,在计算机科学、有机化学、电网络分析、最短连接及渠道设计等领域都有广泛的应用.在本节及下一节中,我们假设所讨论的图与网络都是无向的.

定义 3.1 无圈连通图称为树(tree).无圈图(即若干棵树的并)称为森林或者简称林(forest).如果一个图的支撑子图是一棵树,则称这棵树为该图的支撑树或者生成树(spanning tree),并称支撑树中的弧为树弧(tree arc),而不属于支撑树中的弧为非树弧(nontree arc).

例 3.1 含 6 个节点的树(非同构的)共有 6 种(见图 3.1).所谓两个图是“非同构的”,可以理解为我们不可能通过对其中一个图的节点和弧重新编号(或标号)而使之与另一个图

完全相同.从图形直观上看可以理解为:在平面上经过平移、旋转和弧线长度的伸缩等简单的变换,这两个图不可能重合在一起.否则的话,这两个图就称为是"同构图"(isomorphic graphs).读者可以尝试用数学语言给出图的同构和非同构的严格定义. □

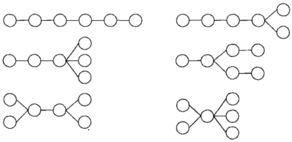


图3.1　由6个节点构成的树

引理3.1　设$G=(N,E)$为一个图,$|N|=n$,则下列命题等价:

(1) G是一棵树;

(2) G连通,且$|E|=n-1$;

(3) G无圈,且$|E|=n-1$;

(4) G的任何两个节点之间存在唯一的一条路;

(5) G连通,且将G的任何一条弧删去之后,该图成为非连通图;

(6) G无圈,且在G的任何两个不相邻节点之间加入一条弧之后,该图正好含有一个圈. □

引理3.1实际上给出了树的一些主要性质.这一引理的证明几乎可以在任何一本有关图论的书中找到,这里我们留作练习,请读者自己给出证明.

定义3.2　$G=(N,E,W)$为一个无向网络,W为权函数,即$W: E\to\mathbb{R}$(这里$\mathbb{R}$表示实数集合).一般将弧$e=(i,j)$上的权记为$W(e)$,W_e或$w(e)$,w_{ij}等.G中权最小(大)的弧称为最小(大)弧.如果$H=(N_1,E_1)$为G的一个子图,子图H的权定义为H所包含的所有弧的权之和,记为$W(H)$,即$W(H)=\sum_{e\in E_1}W(e)$.如果T^*是G的一棵生成树,且对G的任何生成树T,有$W(T^*)\leqslant W(T)$,则称T^*为网络G的最小支撑树或者最小生成树(minimum spanning tree, MST),或者简称最小树.

下面看一个可以用最小树问题来建模的实际问题的例子.

例3.2　数据存储问题

为了节省计算机存储空间,有时候我们希望以更有效的方式存储二维矩阵,而不是直接存储该矩阵的全部元素.例如,某些蛋白质的氨基酸序列差异不多,如果用二维矩阵的每一行记录一种蛋白质氨基酸序列,行与行之间的差异很小.其中一种方法是只存储其中一行作为参照行,再存储行与行之间的一部分差异信息,使得我们可以在需要时根据参照行生成所

有其他行的元素.例如,假定该矩阵一共有 4 行:$R1,R2,R3,R4$,第 i 行与第 j 行的差异为 c_{ij} 个元素.一种可能的存储方式是:只存储 $R1$ 行,以及 $R1$ 与 $R2$、$R1$ 与 $R3$ 和 $R2$ 与 $R4$ 之间的差异元素,则可以根据 $R1$ 和 c_{12} 个差异元素生成 $R2$ 的所有元素;根据 $R1$ 和 c_{13} 个差异元素生成 $R3$;再根据已经得到的 $R2$ 和 c_{24} 个差异元素生成 $R4$.一般地,给定 c_{ij},如何确定存储哪些行之间的差异元素,使得存储空间尽可能少呢?这一问题可以用最小树问题来描述:我们把矩阵每行作为一个节点构成一个完全图,第 i 个节点对应于矩阵第 i 行,并令弧(i,j)上的权为 c_{ij}.对于存储问题,实际上只需要存储一行的元素,以及由该完全图的一棵支撑树所对应的差异元素.最小树就对应于最优的存储方案. □

由前面引理 3.1 知道:支撑树加上一条非树弧后包含一个唯一的圈;支撑树删去一条树弧后形成两棵子树,图中两个端点分属于两棵子树的弧形成一个割.下面我们介绍两个最小树的充要条件,一个是所谓的"割最优条件",另一个是所谓的"圈最优条件".当然,这两个条件本身也是等价的.

定理 3.1 生成树 T^* 是最小树的充要条件是:对 T^* 中的任何一条弧,将该弧从 T^* 中删除后形成的割中,该弧为最小弧.

证明 必要性很容易用反证法证明.设 T^* 是最小树,对 T^* 中的任何一条弧 e,将该弧从 T^* 中删除后形成的割中,如果该弧不是最小弧,设最小弧为 e'.此时,$T^*+e'-e$ 也是生成树,但它的权比 T^* 更小,这与 T^* 为最小树矛盾.因此,弧 e 是割中的最小弧.

充分性:设 T^* 是生成树并满足定理中的条件,但不是最小树.设最小树为 T^0,那么 T^* 中至少有一条弧$(i,j)\notin T^0$.将(i,j)从 T^* 中删除后产生一个割,记为$[S,\bar{S}]$,如图 3.2 所示.将(i,j)加入 T^0 后必然产生一个圈,该圈必然包括一条与(i,j)不同的弧$(k,l)\in[S,\bar{S}]$.根据定理的条件,$w_{ij}\leqslant w_{kl}$;又由于 T^0 为最小树,所以 $w_{kl}\leqslant w_{ij}$,于是只能有 $w_{ij}=w_{kl}$.因此,$T^0+(i,j)-(k,l)$也是最小树,并与 T^* 有更多的公共弧.重复这一过程,最后可以将最小树 T^0 变为 T^*.因此 T^* 也是最小树. □

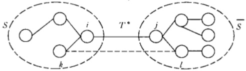


图 3.2 最小树充要条件的证明(定理 3.1 和定理 3.2)

定理 3.2 生成树 T^* 是最小树的充要条件是:对属于 G 但不属于 T^* 的任何一条弧,将该弧加入 T^* 中后形成的圈中,该弧为最大弧.

证明 与上面定理的证明类似,必要性很容易用反证法证明.设 T^* 是最小树,对属于 G 但不属于 T^* 的任何一条弧 e,将该弧加入 T^* 中后形成的圈中,如果该弧不是最大弧,设最大弧为 e'.此时,T^*+e-e' 也是生成树,但它的权比 T^* 更小,这与 T^* 为最小树矛盾.因此,弧 e 是圈中的最大弧.

充分性：设 T^* 是生成树并满足定理 3.2 中的条件，我们来证明它也满足定理 3.1 中的条件. 对任意一条弧 $(i,j)\in T^*$，将 (i,j) 从 T^* 中删除后产生一个割，记为 $[S,\bar{S}]$，并设 $i\in S$，$j\in\bar{S}$. 考虑割中任何一条与 (i,j) 不同的弧 $(k,l)\in[S,\bar{S}]$，则 $(k,l)\notin T^*$. 根据本定理的条件，$w_{ij}\leqslant w_{kl}$，即定理 3.1 中的条件也成立，于是 T^* 是最小树. □

一般称定理 3.1 为"割最优条件"，定理 3.2 为"圈最优条件".

3.2 最小树算法

本节介绍求解最小树问题的几个重要算法，其理论基础就是 3.1 节介绍的"割最优条件"和"圈最优条件". 我们所要介绍的 3 种算法有一个共同的特点：算法开始时假设某支撑子图 T 的弧集合为空集，算法运行过程中不断将一些弧加入到子图中，并且每次加入 T 中的弧就会成为最后找到的最小树的一员，而不会再从 T 中退出. 具有这种特点的算法称为贪婪算法. 也就是说，本节中所要介绍的算法都属于贪婪算法.

本节介绍的算法(或者稍作变形)可以用于其他一些相关的问题. 例如，如果算法结束时无法得到支撑树，则说明图不连通，因为连通图一定有支撑树. 所以这些算法也可以用来判别图是否连通. 同样，既然这些算法可以计算最小树，自然也可以计算连通图的生成树. 并且，利用类似的思路，也很容易设计最大树问题的贪婪算法等.

3.2.1 Kruskal 算法

Kruskal 算法由 Kruskal 于 1956 年提出，其基本思想是每次将一条权最小的弧加入子图 T 中，并保证不形成圈. 也就是说，如果当前弧加入后不形成圈，则加入这条弧，如果当前弧加入后会形成圈，则不加入这条弧，并考虑下一条弧. 当子图中弧的条数为 $n-1$ 时，就得到了最小树. 具体步骤如下：

Kruskal 算法

STEP0 把 G 的弧按照权由小到大的顺序排列，即 $w(e_1)\leqslant w(e_2)\leqslant\cdots\leqslant w(e_m)$；令 $i=1$，$j=0$，$T=\varnothing$.

STEP1 判断 $T\cup e_i$ 是否含圈. 若含圈，转 STEP2，否则转 STEP3.

STEP2 令 $i=i+1$. 若 $i\leqslant m$，转 STEP1；否则结束，此时 G 不连通，所以没有最小树.

STEP3 令 $T=T\cup e_i$，$j=j+1$. 若 $j=n-1$，结束，T 是最小树；否则转 STEP1.

根据定理 3.2，很容易证明 Kruskal 算法的正确性. 下面给出一个数值计算的例子.

例 3.3 用 Kruskal 算法计算如图 3.3(a)所示的网络的最小树.

首先对各条弧按照权由小到大的顺序排列：(1,2)，(4,5)，(1,4)，(2,5)，(3,5)，(2,3)，

(1,3),(3,4).

然后按顺序将这些弧加入 T 中：顺序加入(1,2),(4,5),(1,4)时都不会形成圈，但进一步加入(2,5)时形成圈，因此(2,5)不加入 T 中；进一步加入(3,5)时不形成圈，此时 T 中已经包含 4 条弧，实际上已经得到最小树(如图 3.3 (b))，最小树的权为 11. □

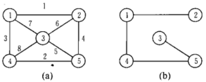


图 3.3 计算最小树的例子

下面分析该算法的计算复杂性. Kruskal 算法的运行时间主要由两部分组成：一是对 m 条弧进行排序，其复杂度为 $O(m\log m)=O(m\log n)$；二是判断加入一条弧到 T 中时是否形成圈，其复杂度依赖于算法的具体实现方法. 可以看出，在算法的计算过程中，T 是一个森林. 如果该森林的每一棵子树中的节点用一个单向链表表示，则可以方便地判断圈. 开始时共有 n 个单向链表，每个链表对应于一个节点(即只含有一个节点). 当考虑弧 (i,j) 时，如果其端点属于同一单向链表，则 (i,j) 加入 T 后会形成圈；如果其端点分属于两个不同的单向链表，则 (i,j) 加入 T 后不会形成圈，因此将这两个链表合并成一个链表. 由于处理每一条弧所需要的时间为 $O(n)$，因此这种实现方法的复杂度为 $O(nm)$.

上面的复杂度可以很容易得到改进. 我们增加 3 个数组：一个是 $size$，用来记录每个链表中所含节点的个数(链表规模)；另一个是 $last$，用来记录每个链表中最后的节点编号；还有一个是 $first$，用来记录每个节点所在链表的第 1 个节点. 例如，如果链表 $L=\{1,2,4,5\}$，则 $size(L)=|L|=4$，$last(L)=5$，$first(1)=first(2)=first(4)=first(5)=1$. 此时，考虑弧 (i,j) 时，只需判断 $first(i)$ 与 $first(j)$ 是否相等，相等则端点 i,j 属于同一单向链表. 因此所有这些判断所需要的计算时间为 $O(m)$. 如果弧 (i,j) 端点属于不同的单向链表，则合并之. 我们总是把节点数较多的链表 L' 放在前面，而把节点数较少的链表 L 追加在后面. 合并后，对于 $size$ 和 $last$ 的修改非常容易，可以在常数时间内完成；但对 $first$ 的修改必须对链表 L 中的每个元素(节点)进行，复杂度为 $O(h)$，其中 $h=size(L)$. 由于这种合并最多进行 $n-1$ 次，下面我们证明对 $first$ 进行修改的总的复杂度为 $O(n\log n)$.

记链表 L 追加在链表 L' $(size(L)=h, size(L')=h', h\leqslant h')$ 后面而合并成一个链表时的计算时间(操作次数)不超过 p_1h(这里 p_1 为常数). 我们只需要证明获得一个规模为 n 的链表的计算时间(操作次数)不超过 $p_2 n\log n$(这里 $p_2\geqslant p_1$ 为常数). 采用数学归纳法：当 $n=1$ 时不需要进行任何合并操作，因此结论成立. 当 $n>1$ 时，我们假设结论对规模小于 n 的链表均成立. 设规模为 n 的链表由链表 L 追加在链表 L' $(size(L)=h, size(L')=h', h\leqslant h')$ 后面合并而成，则 $h\leqslant n/2$，$h'=n-h$. 由归纳假设有：

获得链表 L 的计算时间(操作次数)不超过 $p_2h\log h$；

获得链表 L' 的计算时间(操作次数)不超过 $p_2(n-h)\log(n-h)$.

于是，获得规模为 n 的链表的计算时间(操作次数)不超过

$$p_2h\log h+p_2(n-h)\log(n-h)+p_1h$$

$$\leqslant p_2 h\log(n/2)+p_2(n-h)\log n+p_2 h$$
$$=p_2 n\log n.$$

因此，除了对 m 条弧进行排序所需的时间 $O(m\log n)$外，这一实现方法的复杂度为 $O(m+n\log n)$.

例如，对于例 3.3 中图 3.3(a)所示的例子，开始时设定如下 5 个链表：

$L_1=\{1\}, size(L_1)=1,\ last(L_1)=1, first(1)=1$;

$L_2=\{2\}, size(L_2)=1,\ last(L_2)=2, first(2)=2$;

$L_3=\{3\}, size(L_3)=1,\ last(L_3)=3, first(3)=3$;

$L_4=\{4\}, size(L_4)=1,\ last(L_4)=4, first(4)=4$;

$L_5=\{5\}, size(L_5)=1,\ last(L_5)=5, first(5)=5$.

首先考虑弧(1,2)，由于 $first(1)\neq first(2)$，因此将 L_1, L_2 合并成一个链表(如 L_1)：

$L_1=\{1,2\}, size(L_1)=2,\ last(L_1)=2, first(1)=\ first(2)=1$.

下一步考虑弧(4,5)，由于 $first(4)\neq first(5)$，因此将 L_4, L_5 合并成一个链表(如 L_4)：

$L_4=\{4,5\}, size(L_4)=2,\ last(L_4)=5, first(4)=\ first(5)=4$.

下一步考虑弧(1,4)，由于 $first(1)\neq\ first(4)$，因此将 L_1, L_4 合并成一个链表(如 L_1)：

$L_1=\{1,2,4,5\}, size(L_1)=4,\ last(L_1)=5, first(1)=\ first(2)=\ first(4)=\ first(5)=1$.

下一步考虑弧(2,5)，由于 $first(2)=first(5)$，因此继续考虑下一条弧(3,5). 由于 $first(3)=3\neq first(5)=1, size(L_1)>\ size(L_3)$，因此将 L_1, L_3 合并成一个链表 L_1：

$L_1=\{1,2,4,5,3\}, size(L_1)=5,\ last(L_1)=3, first(1)=\ first(2)=\ first(4)=\ first(5)=first(3)=1$.

此时 $size(L_1)=5=n$，所以已经得到了最小树(结果与例 2.3 相同).

Kruskal 算法目前有更有效的实现方法，除了对 m 条弧进行排序所需的时间 $O(m\log n)$外，其计算复杂性为 $O(m\alpha(n,m))$，其中 $\alpha(n,m)$是用 Ackerman 函数(一种随 m,n 快速增长的函数)定义的某种"反函数"，它随 m,n 的增长非常缓慢，实用中通常可以认为它小于 6 (如当 $n=65536$ 时，$\alpha(n,m)\leqslant 3$).

3.2.2 Prim 算法

Prim 算法又称"边割法"，由 Prim 于 1957 年提出. 它从网络$G=(N,E,W)$的任何一个节点开始，不断扩展一棵子树 $T=(S,E0)$，直到 S 包括原网络的全部节点，得到最小树 T. 具体来说，就是在 $E0$ 中每次增加一条弧，使得这条弧是由当前子树节点集 S 及其补集所形成的边割$[S,\bar{S}]$的最小弧. 可见，Prim 算法与 Kruskal 算法一样也是一种贪婪算法，即加入 T 中的弧不再从 T 中退出. 但它与 Kruskal 算法不同的是，在算法运行过程中 Prim 算法只保持一棵子树 T，而 Kruskal 算法则保持的是一个森林，即多棵子树. Prim 算法的具体步骤如下：

Prim 算法

STEP0 设 v 为 N 的任意一个节点，令 $S=\{v\}, E0=\varnothing$.

STEP1 若 $S=N$，结束，$T=(S,E0)$ 为最小树；否则转 STEP2.

STEP2 若 $[S,\bar{S}]=\varnothing$，则 G 不连通，结束；否则，设 $w(e^*)=\min\limits_{e\in[S,\bar{S}]} w(e)$，其中 $e^*=(v1, v2)$，$v1\in S, v2\in\bar{S}$.

令 $S=S\cup\{v2\}, E0=E0\cup\{e^*\}$，转 STEP1.

根据定理 3.1，很容易证明 Prim 算法的正确性. 下面给出一个数值计算的例子.

例 3.4 用 Prim 算法计算图 3.3(a)所示的网络的最小树：选节点 1 为起始节点，即令 $S=\{1\}$，则$[S,\bar{S}]$中的最小弧为(1,2)，首先加入 T 中. 此时 $S=\{1,2\}$，$[S,\bar{S}]$中的最小弧为(1,4)，将它加入 T 中. 此时 $S=\{1,2,4\}$，$[S,\bar{S}]$中的最小弧为(4,5)，将它加入 T 中. 此时 $S=\{1,2,4,5\}$，$[S,\bar{S}]$中的最小弧为(3,5)，将它加入 T 中. 此时已经得到最小树，可以看出与用 Kruskal 算法得到的最小树相同，只是各条弧加入 T 的先后顺序不一样. □

下面分析其计算复杂性. 该算法的主要计算量在于 STEP2(寻找边割中的最小弧). 如果通过对每条弧进行检查寻找边割中的最小弧，则该步的计算复杂度为 $O(m)$. 因为 STEP2 最多执行 $n-1$ 步，所以该算法的复杂度为 $O(mn)$.

下面介绍一种特殊形式的 Prim 算法，这种方法通常又称 Dijkstra 算法(1959 年). 其基本思想是：对于 $\bar{S}$ 中每一个节点 v，赋予两个数值(通常称为"标号")：一个是距离标号 $d(v)$，记录的是节点 v 到集合 S 的"距离"，即边割$[S,\bar{S}]$中以 v 为一个端点的最小弧的费用；另一个是前趋标号 $p(v)$，记录的是边割$[S,\bar{S}]$中以 v 为端点的最小弧的另一个端点. 有了这两个标号，寻找边割中的最小弧就变得容易了，即只需要计算 $j^*=\arg\min\{d(j)\mid j\in\bar{S}\}$(这里 arg 表示取到最小值时所对应的参数 j 的值). 这时，需要加入 $E0$ 中的弧是$(p(j^*),j^*)$，并将 j^* 从 $\bar{S}$ 删除，加入到 S 中. 然后，对于 $\bar{S}$ 中每一个节点 v 修改标号，但注意到只有当 v 与节点 j^* 相邻时，其标号才可能变更，所以这样的节点 v 才需要修改标号. Dijkstra 算法步骤如下：

Dijkstra 算法

STEP0 设 $N=\{1,2,\cdots,n\}$，并记弧(i,j)上的权为 w_{ij}. 令 $S=\{1\}, E0=\varnothing, d_j=w_{1j}$.

STEP1 选取 $d_{j^*}=\min\{d_j\mid j\in\bar{S}\}=w_{ij^*}$ $(i\in S, j^*\in\bar{S})$，若找不到这样的节点，则 G 不连通，结束；否则令 $S=S\cup\{j^*\}, E0=E0\cup\{(i,j^*)\}$.

STEP2 若 $S=N$，结束，$T=(S,E0)$ 为最小树；否则令 $d_j=\min\{d_j, w_{j^*j}\}, j\in\bar{S}$，转 STEP1.

Dijkstra 算法的计算复杂性：第 1 次执行 STEP1 需要 $n-2$ 次比较，第 2 次执行 STEP1

需要 $n-3$ 次比较，依此类推. 同样，第 1 次执行 STEP2 需要 $n-2$ 次比较，第 2 次执行 STEP2 需要 $n-3$ 次比较，依此类推. 因此，Dijkstra 算法的复杂度为 $O(n^2)$.

利用各种形式的堆(heap)，可以得到 Prim 或 Dijkstra 算法的更有效的实现方法，其复杂度可以降为 $O(m\log n)$，$O(m+n\log n)$ 或 $O(m\log\log C)$，等等(这里 C 为网络 G 中最大弧(按绝对值)的费用值).

3.2.3 Sollin 算法

Sollin 算法是由 Sollin 于 1961 年提出的，它可以看成是前面介绍的两种算法的综合. 在迭代过程中，该算法与 Kruskal 算法一样维持多棵互不相交的子树(森林)，并与 Prim 算法类似地在每次迭代中尝试向每棵子树增加一条弧. 具体如下：

Sollin 算法

STEP0 对 N 中任意一个节点 $i\in N$，定义 $N_i=\{i\}$，$T^*=\varnothing$.

STEP1 如果 T^* 中已经有 $n-1$ 条弧，则 T^* 为 G 的最小树；否则，对 T^* 中的所有子树节点集合 N_k，计算边割 $[N_k,\overline{N_k}]$ 中的最小弧，即计算 $w(e_k^*)=\min\limits_{e\in[N_k,\overline{N_k}]} w(e)$，其中 $e_k^*=(i_k,j_k)$，$i_k\in N_k$，$j_k\in\overline{N_k}$. 然后继续 STEP2.

STEP2 对 T^* 中的所有子树节点集合 N_k 及最小弧 $e_k^*=(i_k,j_k)$，$i_k\in N_k$，$j_k\in\overline{N_k}$，将该子树与 j_k 所在的子树合并成一棵子树，并令 $T^*=T^*\cup\{(i_k,j_k)\}$. 然后转 STEP1.

与 Prim 算法类似，根据定理 3.1，很容易证明 Sollin 算法的正确性. 下面给出一个数值计算的例子.

例 3.5 用 Sollin 算法计算图 3.3(a)所示的网络的最小树：开始时 $N_i=\{i\}$，则 $[N_1,\overline{N_1}]$ 和 $[N_2,\overline{N_2}]$ 中的最小弧均为(1,2)，$[N_3,\overline{N_3}]$ 中的最小弧为(3,5)，$[N_4,\overline{N_4}]$ 和 $[N_5,\overline{N_5}]$ 中的最小弧均为(4,5). 首先应当合并 N_1 和 N_2，即 $N_1=\{1,2\}$，并将弧(1,2)加入 T^* 中；然后合并 N_3 和 N_5，并进一步与 N_4 合并，即 $N_3=\{3,5,4\}$，并将弧(3,5)和(4,5)加入 T^* 中. 此时只有两棵子树节点集 N_1 和 N_3，$[N_1,\overline{N_1}]$ 和 $[N_3,\overline{N_3}]$ 中的最小弧均为(1,4). 合并 N_1 和 N_3，将弧(1,4)加入 T^* 中. 此时已经得到最小树，可以看出与用 Kruskal 和 Prim 算法得到的最小树相同，只是各条弧加入的先后顺序不一样. □

Sollin 算法的计算复杂度：该算法的主要计算量在于 STEP1(寻找边割中的最小弧)和 STEP2(子树合并)的循环交替迭代. 由于每次循环迭代时，每棵树都会合并成一棵较大的子树，因此每次循环迭代都会使子树的数量至少减少一半. 所以，循环迭代的总次数为 $O(\log n)$. 下面对每次循环迭代所需要的计算时间进行分析.

我们对网络中的每个节点赋予一个标号，使得当且仅当两个节点属于同一棵子树时，它

们的标号相同(开始时节点 i 赋予标号 i),此时我们可以方便地判断两个节点是否属于同一子树.进一步,我们可以把每一棵子树中的所有节点存储为一个双向循环链表,此时从该子树的任何一个节点开始,可以方便地访问该子树的所有节点.因此,STEP1 最多对每条弧进行两次检查,该步的计算复杂度为 $O(m)$.在 STEP2 中,只需要根据 STEP1 找到的最小弧进行链表合并,并修改节点的标号,复杂度为 $O(n)$.

综上所述,该算法总的复杂度为 $O((m+n)\log n)=O(m\log n)$.

3.3 最小树形图

在前面两节关于最小树的讨论中,我们假设考虑的是无向网络,而有时候我们需要在有向网络上考虑类似的问题.首先看下面的例子.

例 3.6 一家跨国公司的总经理需要向该公司位于不同国家的一些部门经理传达一条重要信息.假设这名总经理可以用以下两种方式之一完成这一信息的传播:第 1 种方式可以称之为"直接方式",即总经理直接给每个部门经理打电话进行传达;第 2 种方式可以称之为"接力方式",即总经理只给某些部门经理打电话,而让这些得到信息的部门经理打电话将信息进一步传达给其他某些部门经理,依此类推,最后将信息传达到所有部门经理.由于不同国家之间的电话费用是不同的,为了节省总的电话费开支,第 1 种方式一般不是最经济的方式.那么在第 2 种方式中,如何决定哪些部门经理应当由总经理直接传达信息,每个得到信息的部门经理又应当给哪些尚未得到信息的部门经理打电话呢?

以每个经理(总经理或部门经理)为节点构造一个有向图,所有可能进行电话通讯的经理之间连一条有向弧,弧上的权表示通讯费用.这时,问题可以转化为有向图上的最小树形图问题. □

定义 3.3 有向图 $G=(V,A)$ 若满足:

(1) 不包含有向圈;

(2) 存在一个节点 i 使得 $d^-(i)=0$(即节点 i 没有入弧),而对其余节点 j 有 $d^-(j)=1$(即节点 j 有且只有一条入弧);则称 G 为以 i 为根(root)的树形图(arborescence),也称有根树(rooted tree)或外向树(out-tree).

定义 3.4 如果一个有向图的支撑子图是一个树形图,则称该树形图为该有向图的支撑树形图(或生成树形图).有向网络中权最小的支撑树形图称为该网络的最小树形图.

例 3.6 中的通讯方式可以用最小树形图来建模,总经理相当于树形图的根.

可以看出,从树形图的根到每一个节点有且仅有一条有向路,我们称这条路的长度为该节点的代(即该节点关于根节点的层数).树形图在去掉每条弧的方向后,得到的无向基本图是一棵树,因此具有 n 个节点的树形图包含 $n-1$ 条弧.

下面介绍求最小树形图的朱-刘算法,它是由中国科学院的朱永津和刘振宏研究员于 1965 年提出的.

定义 3.5 设 v 是网络 N 的任意一个节点,在所有指向 v 的入弧中,权最小的弧称为 v 的最小入弧. 由最小入弧组成的有向圈称为最小入弧圈.

一种自然的想法是,如果在网络 N 中,从每个节点取一条最小入弧,组成的子图是否就是最小树形图? 如果该子图是网络 N 的支撑树形图,则它就是最小树形图. 如果不是,那么是否可以以此子图为基础构造最小树形图? 这就是朱-刘算法的思路. 下面首先介绍相关的一个引理和几个定理.

引理 3.2 在网络 N 中,从每个节点取一条最小入弧,组成的弧集合记为 H,则

(1) 若 $|H|<n-1$,则 N 没有支撑树形图;

(2) 若 $|H|=n-1$,且 H 不含有向圈,则 H 是 N 的最小树形图;

(3) 若 $|H|=n$,设 a 是 H 中权最大的弧,且 $H\backslash a$ 不含有向圈,则 $H\backslash a$ 是 N 的最小树形图. □

上述引理 3.2 不难证明,请读者自己完成.

定理 3.3 设 C 是网络 N 的最小入弧圈,如果 N 存在支撑树形图,则存在 N 的最小树形图 T 满足 $|C\backslash T|=1$.

证明 假设定理不成立,在 N 的最小树形图中找一棵 T 使得 $|C\backslash T|$ 最小. 由于树形图不含有向圈,因此 $|C\backslash T|=K\geqslant 2$. 记 C 中不在 T 上的弧依次为 $a_1=(v_1,v_1')$,$a_2=(v_2,v_2')$,$\cdots$,$a_k=(v_k,v_k')$,则 C 中余下的弧组成的有向路 $P(v_1',v_2)$,$P(v_2',v_3)$,$\cdots$,$P(v_k',v_1)$ 都属于 T. 不妨设 v_1' 是所有 k 个节点 $\{v_1',v_2',\cdots,v_k'\}$ 在 T 中代数最小的节点,则 v_1' 在 T 中不会是 v_2' 的后代. 但由于 v_2 在 T 中是 v_1' 的后代,所以 v_2 在 T 中不会是 v_2' 的后代. 记 a_1' 是 v_2' 在 T 中的入弧,则 $T'=T\cup a_2\backslash a_1'$ 也是 N 的支撑树形图. 由于 a_2 是节点 v_2' 的最小入弧,因此 $W(T')=W(T)+W(a_2)-W(a_1')\leqslant W(T)$(这里 W 表示权函数). 因此 T_1' 也是 N 的最小树形图,且 $|C\backslash T'|=|C\backslash T|-1$,这与 T 的取法矛盾. □

根据这一定理,在构造最小树形图时,如果网络中存在最小入弧圈,可以去掉圈上的一条弧. 如果根节点在圈上,只需去掉圈上的一条弧即可;如果根节点在圈外,必须找一条从圈外指向圈上某节点的弧来代替圈上一条弧. 记圈上所有弧的权和为 $W(C)$,找到的替代弧为 a,圈上与 a 有相同末端的被替代弧为 a_1,则替代后的权和为 $W(C)+W(a)-W(a_1)$,即权和的变化为 $W(a)-W(a_1)$. 一般地,可以引入如下定义.

定义 3.6 设 C 是有向网络 $N=(V,A,W)$ 的一个有向圈,W 为权函数,即 $W:E\to\mathbb{R}$. 记 C 的节点集合为 $V(C)$,记 C 的弧集为 $A(C)$. 定义 N 收缩 C 后得到的新网络 $N^*=(V^*,A^*,W^*)$ 为

$$V^*=(V\backslash V(C))\cup\{y\},\text{(节点 } y \text{ 称为收缩 } C \text{ 后的人工节点)}$$

$$A^*=A\backslash A(C),$$

$$W^*(a)=\begin{cases}W(a), & \text{当 } a \text{ 在 } N^* \text{ 中的末端不是 } y,\\ W(a)-W(a_1)+W(a^*), & \text{当 } a \text{ 在 } N^* \text{ 中的末端是 } y.\end{cases}$$

其中 a_1 是 C 中与 a 有相同末端的弧,a^* 是 C 中权最大的弧. 这一过程称为网络的收缩

(condensation)，N^* 称为 N 的收缩网络(condensed network)，可简记为 $N^*=N/C$.

定理 3.4 设 C 是网络 $N=(V,A,W)$ 的最小入弧圈，$N^*=(V^*,A^*,W^*)$ 是 N 收缩 C 后得到的新网络. 如果 T^* 是 N^* 的最小树形图，则 $T^*\cup C$ 包含了 N 的一棵最小树形图.

证明 记 a^* 是 C 中权最大的弧. 首先证明 $T^*\cup C$ 中包含 N 中的一棵权为 $W^*(T^*)+W(C)-W(a^*)$ 的支撑树形图. 分两种情况讨论：

(1) 人工节点 y 在 T^* 中有入弧(记为 a_0，参见图 3.4(a))：设 C 中与 a_0 有相同末端的弧为 a_1，则 $T_1=(T^*\cup C)\backslash a_1$ 为 $T^*\cup C$ 包含的 N 中的唯一支撑树形图. 此时

$$
\begin{aligned}
W(T_1) &= \sum_{a\in T^*\backslash a_0} W(a)+W(a_0)+W(C)-W(a_1)\\
&= \sum_{a\in T^*\backslash a_0} W^*(a)+W^*(a_0)+W(C)-W(a^*)\\
&= W^*(T^*)+W(C)-W(a^*).
\end{aligned}
$$

(2) 人工节点 y 是 T^* 的根节点(参见图 3.4(b))：此时 $T_2=(T^*\cup C)\backslash a^*$ 为 $T^*\cup C$ 包含的 N 中的一棵支撑树形图，并且

$$W(T_2)=W^*(T^*)+W(C)-W(a^*).$$

因此，不论哪种情况，$T^*\cup C$ 都包含 N 中的一棵权为 $W^*(T^*)+W(C)-W(a^*)$ 的支撑树形图.

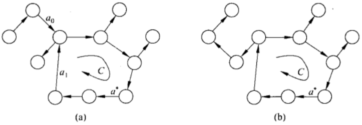


图 3.4 定理 3.4 的证明

根据定理 3.3，存在 N 的最小树形图 T_0 使得 $|C\backslash T_0|=1$. 在收缩 C 后，T_0 变成 N^* 中的支撑树形图 T_0^*，因此 $W(T_0^*)\geqslant W(T^*)$. 类似于上面两种情况的讨论，可以进一步得到

$$W(T_0)=W^*(T_0^*)+W(C)-W(a^*).$$

于是只能有 $W(T_0^*)=W(T^*)$，也就是说 T_0^* 是 N^* 中的最小树形图. 因为否则的话，T_0 不会是 N 中的最小树形图 (T_1 或 T_2 才是 N 中的最小树形图). 因此我们可以得到

$$W(T_0)=W^*(T^*)+W(C)-W(a^*).$$

由此可以看出，不仅 T_0 是 N 中的最小树形图，T_1 或 T_2 也是 N 中的最小树形图. 因此，$T^*\cup C$ 包含了 N 的一棵最小树形图. □

根据上述定理，可以设计如下最小树形图算法：

朱-刘算法

STEP0 令 $m=1, N_m(V_m, A_m, W_m)=N(V,A,W)$.

STEP1 在 N_m 中对每个节点取一条最小入弧，组成的弧集记为 H_m. 若 $|H_m|<|V_m|-1$，则原网络没有支撑树形图；否则若 $|H_m|=|V_m|$，则去掉 H_m 中的权最大的一条弧（仍记为 H_m），继续下一步；否则直接继续下一步.

STEP2 若 H_m 不包含圈，则令 $H'_m=H_m$，转 STEP4；否则取 H_m 包含的一个圈 C_m，继续下一步.

STEP3 对 N_m 收缩 C_m 得到新的网络 $N_{m+1}(V_{m+1}, A_{m+1}, W_{m+1})$，记人工节点为 y_m. 令 $m=m+1$，转 STEP1.

STEP4 若 $m=1$，则 H'_1 就是 N 中的最小树形图，结束；否则继续下一步.

STEP5 令 $H'_{m-1}=H'_m\cup(C_{m-1}\backslash a'_{m-1})$，其中 a'_{m-1} 是 C_{m-1} 中的一条弧：如果 y_{m-1} 在 H'_m 中有入弧，则 a'_{m-1} 取为与该入弧有相同末端的弧；否则 a'_{m-1} 取 C_{m-1} 中的权最大的一条弧. 令 $m=m-1$，转 STEP4.

上述算法实际上包括两大过程：收缩(STEP1～STEP3)和展开(STEP4～STEP5). 每一个过程最多循环 $n-1$ 次. 容易看出 STEP1 的复杂度为 $O(m)$，STEP2 的复杂度为 $O(n)$，STEP3 的复杂度为 $O(m)$，STEP4 的复杂度为 $O(1)$，STEP5 的复杂度为 $O(m)$，因此朱-刘算法的总复杂度为 $O(mn)$.

该算法同样可以用于计算最大树形图（只需将权反号即可）. 对算法作一些小的改动，容易设计求具有固定根的最小树形图或最大树形图（留作练习）.

例 3.7 求图 3.5(a)中网络 N 的最小树形图

计算过程如下：

记 $N_1=N$，对每个节点取一条最小入弧. 由于每个节点都存在最小入弧，因此进一步去掉其中权最大的一条弧(5,4)，组成的弧集为 H_1（图 3.5(b)）.

N_1 收缩有向圈 $C_1=\{(3,5),(5,6),(6,3)\}$，得到网络 N_2（图 3.5(c)）.

对每个节点取一条最小入弧. 由于每个节点都存在最小入弧，因此进一步去掉其中权最大的一条弧$(y_1,4)$，组成的弧集为 H_2（图 3.5(d)）.

N_2 收缩有向圈 $C_2=\{(1,2),(2,y_1),(y_1,1)\}$，得到网络 N_3（图 3.5(e)）.

对每个节点取一条最小入弧. 由于每个节点都存在最小入弧，因此进一步去掉其中权最大的一条弧$(y_2,4)$，组成的弧集为 H_3（图 3.5(f)）.

由于 H_3 不含圈，故收缩过程结束，令 $H'_3=H_3$. 展开得到 H'_2（图 3.5(g)）.

进一步展开得到 H'_1（图 3.5(h)），这就是最小树形图. □

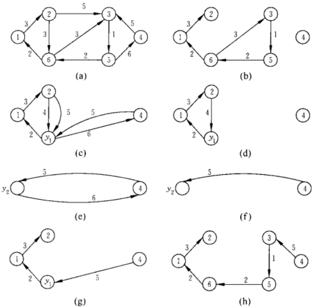


图 3.5 计算最小树形图的例子(例 3.7)

3.4 最大分枝

支撑树形图是有向连通图. 有时候,我们并不需要在有向图中找到支撑树形图,而只需要将有向图分解成一些树形图的并. 图论中一般把多个树形图的并所构成的图形称为分枝. 因此本节我们简要介绍一下最大分枝问题及其算法.

定义 3.7 若有向图 G 满足定义 3.3 中的条件(1),且满足(2)对任意节点 j 有 $d^-(j)\leqslant 1$, 则称 G 为分枝(branching).

可见,分枝是若干树形图的并,正如森林是若干树的并一样. 树形图本身也是分枝的一种特殊情况(即由一个树形图组成的分枝).

定义 3.8 如果一个有向图的生成子图是一个分枝,则称该分枝为该有向图的支撑分枝(或生成分枝). 有向网络中权最大的支撑分枝称为该网络的最大分枝.

显然,我们可以断言: 最大分枝中不会含有负权弧. 但是,即使网络 N 中所有弧上的权均为正数,并且该网络 N 存在支撑树形图,N 的最大分枝也不一定是支撑树形图(自然更谈不上是最大树形图). 例如下面的例子可以说明这一点(图 3.6).

下面介绍 Edmons 于 1968 年提出的最大分枝算法,它与求最小树形图的朱-刘算法本

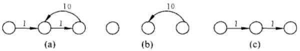


图 3.6　最大分枝的例子

(a) 网络 N；(b) N 的最大分枝；(c) N 的最大树形图

质上是一样的，因此我们这里直接给出算法，证明和复杂度分析留作练习.

Edmons 算法

STEP0　令 $m=1, N_m(V_m, A_m, W_m)=N(V, A, W)$.

STEP1　在 N_m 中对每个节点取一条最大正权入弧，组成的弧集记为 B_m.

STEP2　若 B_m 不包含圈，则令 $B'_m=B_m$，转 STEP4；否则取 B_m 包含的一个圈 C_m，继续下一步.

STEP3　对 N_m 收缩 C_m 得到新的网络 $N_{m+1}(V_{m+1}, A_{m+1}, W_{m+1})$，记人工节点为 y_m. 令 $m=m+1$，转 STEP1.

STEP4　若 $m=1$，则 B'_1 就是 N 中的最大分枝，结束；否则继续下一步.

STEP5　令 $B'_{m-1}=B'_m \cup (C_{m-1} \backslash a'_{m-1})$，其中 a'_{m-1} 是 C_{m-1} 中的一条弧：如果 y_{m-1} 在 B'_m 中有入弧，则 a'_{m-1} 取为与该入弧有相同末端的弧；否则 a'_{m-1} 取 C_{m-1} 中的权最小的一条弧. 令 $m=m-1$，转 STEP4.

值得注意的是：由于此时是计算"最大"分枝，因此在 STEP3 中执行收缩时，收缩后网络中权的定义与 3.3 节中的定义会略有不同：即将定义 3.6 中的"a^* 是 C 中权最大的弧"相应地改为"a^* 是 C 中权最小的弧".

例 3.8　求如图 3.7(a)中网络 N 的最大分枝.

计算过程如下：

记 $N_1=N$，对每个节点取一条最大正权入弧，组成的弧集为 B_1(图 3.7(b)).

N_1 收缩有向圈 $C_1=\{(1,2),(2,4),(4,1)\}$，得到网络 N_2(图 3.7(c)).

对每个节点取一条最大正权入弧，组成的弧集为 B_2(图 3.7(d)).

N_2 收缩有向圈 $C_2=\{(y_1,5),(5,y_1)\}$，得到网络 N_3(图 3.7(e)).

对每个节点取一条最大正权入弧，组成的弧集为 B_3(图 3.7(f)).

由于 B_3 不含圈，故收缩过程结束，令 $B'_3=B_3$. 展开得到 B'_2 (图 3.7(g)).

进一步展开得到 B'_1 (图 3.7(h))，这就是最大分枝.　□

显然，这一算法可以用于计算最小分枝(弧上的权反号即可). 此外，如果对网络进行一定的预处理，这一算法可以用于计算最小(或最大)树形图，以及具有固定根的最小(或最

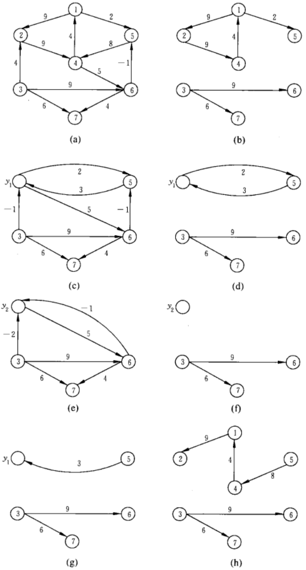


图 3.7 计算最大分枝的例子(例 3.8)

大)树形图.我们这里作为练习留给读者自己完成.

练 习 题

1. 证明引理 3.1.

2. 设 T 是无向网络 $N=(V,A,W)$ 的支撑树，证明：T 是网络 N 的唯一最小树的充要条件是如下两个条件之一成立：

(1) 对一切非树弧 e，e 是 $T+e$ 所含圈中唯一权最大的弧.

(2) 对一切树弧 e，e 是 $T\backslash e$ 所形成的割中唯一权最小的弧.

3. 分别用本章介绍的 3 种最小树算法计算图 3.8 中网络的最小树.

4. 分别用本章介绍的 3 种最小树算法计算图 3.9 中网络的最小树 .

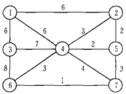


图 3.8 第 3 题图

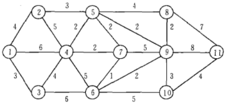


图 3.9 第 4 题图

5. 计算题 4 网络中包含与节点 1 相关的 3 条弧(1,2),(1,3)和(1,4)的权最小的一棵支撑树.

6. 无向网络的一棵支撑树加上一条非树弧，所形成的子图称为该网络的一棵 1-树(又称单圈子图). 试设计一个算法，计算无向网络的最小 1-树，并使得该 1-树中唯一的圈通过一个指定节点. 分析算法的计算复杂度.

7. 在无向网络的所有支撑树中，最大弧与最小弧的权之差最小的一棵称为平衡树. 试设计一个算法，计算无向网络的平衡树，并分析算法的计算复杂度.

8. 在无向网络的所有支撑树中，使得指定节点的度数为 k 的一棵称为 k 度限制树. 试设计一个算法，计算无向网络的 k 度限制树，并分析算法的计算复杂度.

9. 将无向网络的所有支撑树按照权非降的顺序排列，称处于第 k 个位置的支撑树为第 k 个支撑树. 试设计一个算法，计算无向网络的第 2 个支撑树，并分析算法的计算复杂度.

10. 在无向网络中，如果去掉一条弧后使得最小树的权严格增加，则称该弧为"活跃弧". 去掉后使得最小树的权增加最多的一条弧称为"最活跃弧". 试设计一个算法，计算无向网络的"最活跃弧"，并分析算法的计算复杂度.

11. 考虑具有两个权函数的无向网络：每条弧(i,j)上赋予权 w_{ij} 和 c_{ij}，并设对于该无向网络的任意支撑树 T，有 $\sum_{(i,j)\in T} c_{ij}>0$. 试设计一个算法，计算该网络的最小比例支撑树，即

使得 $\sum_{(i,j)\in T} w_{ij} \Big/ \sum_{(i,j)\in T} c_{ij}$ 最小的支撑树，并分析算法的计算复杂度.

12. 最小树的敏感性分析：设计一个算法，对无向网络的每条弧上的权计算一个区间，使得当权在该区间内变化时，最小树不变. 分析算法的计算复杂度.

13. 对朱-刘算法作一些小的改动，设计求具有固定根的最小树形图.

14. 计算如图 3.10 所示网络的最小树形图.

15. 计算题 14 中网络的以节点 1 为根的最小树形图.

16. 计算题 14 中网络的最大树形图.

17. 计算题 14 中网络的最大分枝.

18. 计算如图 3.11 所示网络的最大分枝.

19. 证明 Edmons 算法的正确性，并分析其计算复杂度.

20. 对网络进行一定的预处理，将 Edmons 算法推广到用于计算最小(或最大)树形图，以及具有固定根的最小(或最大)树形图.

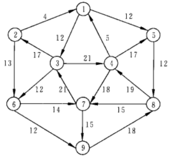


图 3.10 第 14 题图

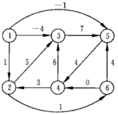


图 3.11 第 18 题图

第 4 章　最短路问题

在例 1.2 中介绍了最短路问题，这一章将进一步详细介绍最短路问题及其有效算法.

最短路问题是网络优化中的一个相对基本而又非常重要的问题. 首先，这一问题在实际生产和生活中经常会遇到，许多实际问题都可以转化为最短路问题. 其次，这一问题相对比较简单，其有效算法经常在其他网络优化问题中作为子算法调用.

4.1　最短路问题的数学描述

最短路问题考虑的是有向网络$N=(V,A,W)$，其中弧$(i,j)\in A$对应的权 w_{ij} 又称为弧长或费用. 对于其中的两个节点 $s,t\in V$，以 s 为起点 t 为终点的有向路称为 s-t 有向路，其所经过的所有弧上的权(或弧长、费用)之和称为该有向路的权(或弧长、费用). 所有 s-t 有向路中权最小的一条称为 s-t 最短路. 同样道理，可以定义最长路问题. 最长路问题可以方便地转化为最短路问题，这只需把弧上的费用反号即可.

对于有向网络中的一个圈，它的权就是圈上所有前向弧上的权的和，减去圈上所有反向弧上的权. 权为正的圈称为正圈；权为负的圈称为负圈；权为 0 的圈称为零圈. 对于一个有向圈，它的权就是圈上所有弧上的权的和. 由于本章一般讨论有向网络，讨论的圈一般都是指有向圈，所以本章后面直接将正有向圈简称为"正圈"，负有向圈简称为"负圈"，零有向圈简称为"零圈"，而"无圈"指的就是不存在有向圈.

必须指出的是：目前为止，一切最短路算法都只对不含负有向圈的网络有效. 实际上，对于含负有向圈的网络，其最短路问题是 NP 困难的. 因此，本章中除非特别说明，一律假定网络不包含负有向圈. 此外，在实际问题中有时也会遇到无向网络上的最短路问题，这时原问题一般可以转化为有向网络上的最短路问题. 如果所有弧上的权 w_{ij} 全为非负(或非正)数，只需将无向图的一条边代之以两条对称的有向弧即可. 如果弧上的权 w_{ij} 有负有正，一般来说问题要复杂得多，应当具体问题具体分析.

网络不包含负有向圈时，最短路问题可以用线性规划描述如下：

$$\min \sum_{(i,j)\in A} w_{ij}x_{ij} \tag{4.1}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = \begin{cases} 1, & i=s, \\ -1, & i=t, \\ 0, & i\neq s,t. \end{cases} \tag{4.2}$$

$$x_{ij}\geqslant 0. \tag{4.3}$$

其中决策变量 x_{ij} 表示弧(i,j)是否位于 s-t 路上：当 $x_{ij}=1$ 时，表示弧(i,j)位于 s-t 路上，当 $x_{ij}=0$ 时，表示弧(i,j)不在 s-t 路上. 本来，x_{ij} 应当是 0-1 变量，但由于约束条件(4.2)的约束矩阵就是网络的关联矩阵，它是全幺模矩阵(参见 2.2.3 节)，因此 0-1 变量可以松弛为区间[0,1]中的实数(当用单纯形法求解时，将得到 0-1 整数解).

值得注意的是，我们这里将变量 x_{ij} 直接松弛为所有非负实数. 实际上，如果 x_{ij} 可以取 0-1 以外的整数，则约束条件并不能保证对应于非零 x_{ij} 的弧所构成的结构(记为 P)一定是一条路，因为这一结构可能含有圈. 进一步分析，由于我们总是假设网络本身不含有负圈，而任何正圈不可能使目标函数最小，因此上面的约束条件(4.2)，(4.3)可以保证当达到最优解时，P 如果包含圈，该圈一定是零圈. 我们从 P 中去掉所有的零圈，就可以得到最短路.

下面我们介绍几个可以用最短路问题来建模的实际问题.

例 4.1 单产品、无能力限制的批量问题

某工厂生产某种产品用以满足市场需求，且已知在时段 t 中的市场需求为 $d_t(t=1,2,\cdots,T)$. 在某时段 t，如果开工生产，则生产开工所需的生产准备费为 $s_t\geqslant 0$，单件产品的生产费为 $c_t\geqslant 0$. 在某时段 t 期末，如果有产品库存，单件产品的库存费为 $h_t\geqslant 0$. 假设初始库存为 0，不考虑能力限制，工厂应如何安排生产，可以保证按时满足生产，且使总费用最小?

假设在时段 t，产品的生产量为 $x_t(x_t\geqslant 0)$，期末产品的库存为 $I_t(I_t\geqslant 0)$，则由问题的假设知 $I_0=0$. 已得：在最优解中，$\sum\limits_{t=1}^{T}x_t=\sum\limits_{t=1}^{T}d_t$，而且一定存在满足条件 $I_{t-1}x_t=0(1\leqslant t\leqslant T)$ 的最优解. 因此，当产品在某一时段有生产时，可以只考虑其生产量为某些后续时段的需求量的总和，即 $x_t\in\{d_t,d_t+d_{t+1},\cdots,d_t+d_{t+1}+\cdots+d_T\}$. 也就是说，只有当上一时段库存为 0 时，本时段才考虑进行生产. 记 w_{ij} 为第 i 时段生产量为 $x_i=d_i+d_{i+1}+\cdots+d_j$ 时所导致的费用(包括生产准备费、生产费和库存费)，即

$$w_{ij}=s_i+c_ix_i+\sum_{t=i}^{j-1}h_tI_t,$$

其中 $I_t=d_{t+1}+d_{t+2}+\cdots+d_j(i\leqslant t\leqslant j-1)$.

此时，由于该问题的特殊性，可以转化为最短路问题. 构造一个网络，该网络有 $T+1$ 个节点，前 T 个节点对应于 T 个时段，最后一个节点表示生产结束时的对应状态. 从所有节点 i 到 j $(j>i)$连一条弧，弧上的权(费用或长度)为 $w_{i,j-1}$. 此时，从节点 1 到节点 $T+1$ 的任何一条有向路，对应于一个生产计划方案. 如果该有向路包含弧(i,j)，则表示第 i 时段生产量用于满足从 i 到 $j-1$ 时段的需求. 最短路对应的正是最优计划.

例如，当 $T=4$ 时，图 4.1 给出了对应的最短路问题的网络. □

例 4.2 最可靠路径问题

在一个通讯网络中，任意两个中继 i 和 j 之间的线路可靠概率为 $P_{ij}(P_{ij}>0)$. 对于从中继 s 到 t 的任意有向路 $L=si_1i_2\cdots i_kt$(不含有重复节点)，其可靠性为

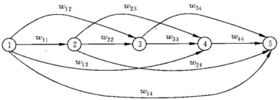


图 4.1 生产批量问题对应的网络

$$M = P_{si_1} P_{i_1 i_2} \cdots P_{i_k t}.$$

为了找到可靠性 M 最大的有向路，上式两边取对数，得

$$\log M = \log P_{si_1} + \log P_{i_1 i_2} + \cdots + \log P_{i_k t}.$$

如果以 $-\log P_{ij}$ 作为中继(节点) i 和 j 之间的弧长，则最可靠路径的问题实际上就是一个最短路问题. □

例 4.3 计划评审技术(project evaluation & review technique，PERT)

PERT 又称网络计划技术或统筹法. 大型复杂工程项目(project)往往被分成许多子项目，子项目之间有一定的先后顺序(偏序)要求，每一子项目需要一定的时间完成 (如图 4.2). PERT 网络的每条弧表示一个子项目，如果以弧长表示每一子项目需要的时间，则最早完工时间对应于网络中的最长路(关键路线). PERT 网络中不能含有任何圈，因此其最长路问题和最短路问题都是可解的. 工程上所谓的关键路线法(critical path method，CPM)基本上也是计划评审技术的一部分. □

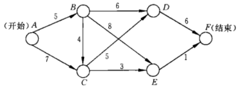


图 4.2 项目网络

4.2 无圈网络与正费用网络：标号设定算法

4.2.1 Bellman 方程

我们可以根据 4.1 节介绍的线性规划(4.1)～(4.3)构造对偶问题. 为了方便后面的讨论，我们先将约束(4.2)两边同时乘以 -1，然后可以得到其对偶问题为

$$\max(u_t - u_s) \tag{4.4}$$

$$\text{s.t.}\quad u_j - u_i \leqslant w_{ij}, \quad \forall (i,j) \in A. \tag{4.5}$$

根据互补松弛条件，当 x 和 u 分别为原问题和对偶问题的最优解时：

$$x_{ij}(u_j - u_i - w_{ij}) = 0, \quad \forall (i,j) \in A. \tag{4.6}$$

因此，当某弧 (i,j) 位于最短路上时，即对应的变量 $x_{ij} > 0$ 时，一定有 $u_j - u_i = w_{ij}$. 但在对偶问题中，如果 u 为最优解，易知 $u+c$ (c 为任意实数)仍为最优解. 因此，u 的具体数值不能唯

一确定. 不妨令 $u_s=0$，则 u 的具体数值就可以唯一确定了.

进一步分析可以看出，u_j 相当于对节点 j 赋予的一个实数值(通常称为"标号")，在 $u_s=0$ 时 u_j 表示的正好是节点 s 到节点 j 的最短路的长度. 因此，可以得到求解最短路问题的如下递推关系：

$$\begin{cases} u_s = 0, \\ u_j = \min\limits_{i\neq j}\{u_i + w_{ij}\}. \end{cases} \tag{4.7}$$

当两节点间没有弧相连接时，上面递推式中认为对应的权为无穷大. 其实，回顾上一章的内容可以看出，这就是最短路问题的动态规划基本方程，称为 Bellman 方程，也称为最短路方程. 但由于求解 u_j 时必须知道所有其他节点 i 上的 u_i 值，因此一般情况下直接求解最短路方程是相当困难的.

定理 4.1 对于只含正有向圈的连通有向网络，从起点 s 到任一节点 j 都存在最短路，它们构成以起点 s 为根的树形图(称为最短路树(tree of shortest paths)或最短路树形图(shortest path arborescence))，最短路的长度可以由 Bellman 方程唯一确定. □

上述定理中的前一部分实际上是 Bellman 最优化原理的直接结果，后一部分中 Bellman 方程的解的唯一性可以用反证法证明，请读者完成. 必须指出的是，如果将定理中的条件"只含正有向圈"改为"不含负有向圈"，则上述最短路仍然存在，但 Bellman 方程的解的唯一性可能不成立. 例如，在如下所示的网络中(图 4.3)，起点 s 到任一节点的最短路的长度分别是 $u_s=0, u_2=10, u_3=11$，但此时只要 $u_3=u_2+1\leqslant 11$ 就可以满足 Bellman 方程. 例如，$u_s=0, u_2=8, u_3=9$ 就满足 Bellman 方程. 由此可以看出，对于一般的网络，Bellman 方程只是最短路长度必须满足的必要条件，而不是充分条件. 对于只含正有向圈的连通有向网络，它才是充要条件.

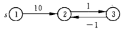


图 4.3 Bellman 方程解不唯一的例子

4.2.2 无圈网络

如果给定的网络是无有向圈的，则定理 4.1 的条件自然满足，并且此时最短路的长度可以方便地求到. 此时，我们可以对网络中的节点进行拓扑排序(例 2.19)，使得 $\forall i\geqslant j, (i,j)\notin A$. 拓扑排序的复杂度为 $O(m+n)$.

如果拓扑排序已经完成，则 Bellman 方程(4.7)可以写成

$$\begin{cases} u_s = u_1 = 0, \\ u_j = \min\limits_{i<j}\{u_i + w_{ij}\}. \end{cases} \tag{4.8}$$

可以看出，此时可以采用递推计算求解上述方程. 当采用递推计算求解上述方程时，每个节点和每条弧均被考虑一次，因此这一递推计算也可以在 $O(m+n)=O(m)$ 的时间内实现. 于是，当网络是无圈时，这一最短路算法的复杂度为 $O(m+n)=O(m)$. 这是求解最短路问题可能达到的最小的复杂度，因为任何算法都至少必须对每条弧考虑一次.

例 4.4　计算如下网络(图 4.4 (a))中从节点 A 到所有其他节点的最短路.

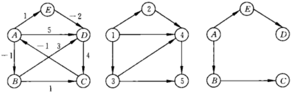


图 4.4　无圈网络的最短路例子(例 4.4)

初步观察，发现网络本身含有有向圈，似乎不是无圈网络. 但是，由于是计算从节点 A 到所有其他节点的最短路，所以节点 A 的入弧可以去掉，得到一个无圈网络. 此时，对网络进行拓扑排序(设 A 点编号为 1)，可以得到如图 4.4(b)的网络.

计算过程：$u_1=0$，

$$u_2=\min_{i<2}\{u_i+w_{i2}\}=\min\{0+1\}=1,$$

$$u_3=\min_{i<3}\{u_i+w_{i3}\}=\min\{0+(-1)\}=-1,$$

$$u_4=\min_{i<4}\{u_i+w_{i4}\}=\min\{0+5,\ 1+(-2),\ -1+3\}=-1,$$

$$u_5=\min_{i<5}\{u_i+w_{i5}\}=\min\{-1+1,\ -1+4\}=0.$$

最后得到的最短路树形图如图 4.4(c)所示. □

4.2.3　正费用网络

下面介绍正费用网络的 Dijkstra 算法(由 Dijkstra 于 1959 年提出). 所谓正费用网络，就是说弧上的所有权为正数. 算法的基本思想是：对于 V 中每一个节点 j，赋予两个数值(通常称为“标号”)：一个是距离标号 u_j，记录的是从起点到该节点的最短路长度的上界；另一个是前趋标号 pred(j)，记录的是当起点 s 到该节点 j 的一条路长取到该上界 u_j 时，该条路中节点 j 前面的那个直接前趋(节点). 算法通过不断修改这些标号，进行迭代计算. 当算法结束时，距离标号表示的是从起点到该节点的最短路长度. 在算法不断修改这些标号、迭代进行计算的过程中，所有节点实际上被分成了两类：一类是离起点 s 较近的节点，它们的距离标号表示的是从点 s 到该节点的最短路长度，因此其标号不会在以后的迭代中再被改变(称为永久标号)；另一类是离起点 s 较远的节点，它们的距离标号表示的只是从点到该节点的最短路长度的上界，因此其标号还可能会在以后的迭代中再被改变(称为临时标号).

算法的具体步骤如下：

Dijkstra 算法(计算正费用网络 $G=(V,A,W)$ 的最短路，起点 s 的编号假定为节点 1)

STEP0　(初始化) 令 $S=\varnothing,\bar{S}=V,u_s=u_1=0,\text{pred}(s)=0$；对 V 中的节点 j $(j\neq s)$令初始

距离标号 $u_j=\infty$.

STEP1 如果 $S=V$，则 u_j 为节点 s 到节点 j 的最短路路长(最短路可以通过数组 pred 所记录的信息反向追踪获得)，结束. 否则继续 STEP2.

STEP2 从 $\overline{S}$ 中找到距离标号最小的节点 i，把它从 $\overline{S}$ 删除，加入 S. 对于所有从 i 出发的弧 $(i,j)\in A$，若 $u_j>u_i+w_{ij}$，则令 $u_j=u_i+w_{ij}$，pred$(j)=i$. 转 STEP1.

下面说明该算法的正确性. 我们证明如下结论：

引理 4.1 在上述 Dijkstra 算法中，

(1) 对于 S 中的任一节点 j，其距离标号是从起点 s 到该节点 j 的最短路路长；

(2) 对于 $\overline{S}$ 中的任一节点 j，其距离标号是从起点 s 出发，只经过 S 中的节点到达节点 j 的最短路路长.

显然，如果以上两个结论成立，则算法终止时，就求到了从起点 s 到任意一个节点 j 的最短路长. 并且，该最短路可以通过前趋标号(pred)求得(回溯法).

证明 我们用数学归纳法证明. 在算法开始时，上述结论显然成立. 现在假设直到迭代的第 k 步，上述结论(1)，(2)成立.

在下一步(第 $k+1$ 次迭代)，$\overline{S}$ 中具有最小标号的节点 i 可以移入 S 中，这不会破坏结论(1). 也就是说，此时 u_i 就是从起点 s 到该节点 i 的最短路路长. 可以如下考虑：设节点 i 是 $\overline{S}$ 中具有最小标号的节点，考察任意一条 s-i 路 P，设 P 经过的 $\overline{S}$ 中的第一个节点为 j，记 $P1$ 是 P 中从起点 s 出发只经过 S 中的某些节点最后到达节点 j 的子路. 即 j 将 P 分为 $P1$ 和 $P2$ 两条子路(如图 4.5)，根据归纳假设(2)成立，可知 $P1$ 的路长为 u_j. 因为 i 是 $\overline{S}$ 中具有最小标号的节点，所以 $u_j\geqslant u_i$；又由于网络中的所有弧费用为正数，所以 P 的路长大于等于 $P1$ 的路长，即网络中任意一条 s-i 路的路长大于等于 $u_j\geqslant u_i$. 也就是说，u_i 是 s-i 最短路，节点 i 可以移入 S 中，不会破坏结论(1).

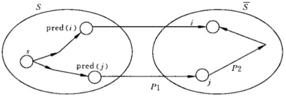


图 4.5 Dijkstra 算法的正确性证明

进一步分析，当节点 i 可以移入 S 中以后，对于 $\overline{S}$ 中的其他节点 j，通过修改标号，不会破坏结论(2). 如果从起点 s 只经过 S 中的节点到达节点 j 的最短路所经过的 S 中的最后一个节点是 i，则节点 j 的标号需要进行算法中所示的修正；否则，节点 j 的标号不变. □

综上所述，该算法是正确的最短路算法.

例 4.5　计算如下网络(图 4.6 (a))中从节点 1 到所有其他节点的最短路.

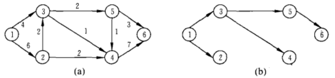


图 4.6　正费用网络的最短路例子(例 4.5)

计算过程(这里没有详细给出 pred 的修改过程，请读者补上)：$V=\{1,2,3,4,5,6\}$

开始时 $S=\varnothing,\bar{S}=V=\{1,2,3,4,5,6\}, u_1=0, u_j=\infty\ (j\neq 1)$.

下一步：$S=\{1\},\bar{S}=\{2,3,4,5,6\}$，考虑从节点 1 出发的弧：

$$u_2=u_1+w_{12}=0+6=6,\quad u_3=u_1+w_{13}=0+4=4.$$

下一步：$S=\{1,3\},\bar{S}=\{2,4,5,6\}$，考虑从节点 3 出发的弧：

$$u_5=u_3+w_{35}=4+2=6,\quad u_4=u_3+w_{34}=4+1=5.$$

下一步：$S=\{1,3,4\},\bar{S}=\{2,5,6\}$，考虑从节点 4 出发的弧：

$$u_6=u_4+w_{46}=5+7=12.$$

下一步：$S=\{1,3,4,2\},\bar{S}=\{5,6\}$，考虑从节点 2 出发的弧：$u_3,u_4$ 不变.

下一步：$S=\{1,3,4,2,5\},\bar{S}=\{6\}$，考虑从节点 5 出发的弧：

$$u_4 \text{ 不变},\quad u_6=u_5+w_{56}=6+3=9.$$

下一步：$S=\{1,3,4,2,5,6\}=V$，最后得到的最短路树形图如图 4.6(b)所示.　□

计算复杂性分析：该算法的主要计算量在于 while 循环(最多执行 n 次)，它包括两个过程：节点寻找过程(从 $\bar{S}$ 中找到距离标号最小的节点 i)和距离修改过程(修改与节点 i 相邻的节点的距离标号).

对于节点寻找过程，第 1 次循环时需要 n 次比较操作，第 2 次循环时需要 $n-1$ 次比较操作，……依此类推. 因此，节点寻找过程的复杂度为 $n+(n-1)+\cdots+1=O(n^2)$.

对于距离修改过程，注意到一个节点只有当它与节点 i 相邻时，其标号才可能变更，才需要修改标号. 每次循环所需要修改标号的节点个数最多为 $d^+(i)$，即节点 i 的出弧数(出度). 因此，这一过程的整体效应相当于对每条弧进行一次检查，所以该步的总计算复杂度为 $O(m)$.

综上所述，该算法的复杂度为 $O(n^2+m)=O(n^2)$.

对于稠密网络($m\approx n^2$)，这是求解最短路问题可能达到的最小的复杂度，因为任何算法都至少必须对每条弧考虑一次. 对于稀疏网络(m 远远小于 n^2)，利用各种形式的堆(heap)，可以得到 Dijkstra 算法更有效的实现方法，其复杂度可以降为 $O(m\log n)$，$O(m+n\log n)$或 $O(m+n\sqrt{\log C})$，等等(这里 C 为网络 G 中最大弧(按绝对值)的费用值).

在网络优化中，一般将这种对节点赋予距离标号(数值)，并通过迭代过程对距离标号进行逐步修正的最短路算法称为标号算法(labeling algorithm). 标号算法中的距离标号可以分为两类：一类表示的已经是最短路的路长，因此在以后的迭代过程中不再改变，称为永久标号；另一类表示的只是最短路路长的估计值(上界)，因此在以后的迭代过程中可能还会不断得到改进，称为临时标号. 标号算法的目的就是在算法结束时将所有临时标号转变为永久标号. 本节中前面介绍的求解无圈网络的最短路算法(即求解方程(4.8)的递推算法)，也可以看成是一种标号算法.

求解最短路问题的标号算法一般可以分成两类，即所谓的标号设定算法(label-setting algorithm)和标号修正算法(label-correcting algorithm). 本节中介绍的两种标号算法有如下的共同特点：在通过迭代过程对标号进行逐步修正的过程中，每次迭代将一个节点从临时标号集合中移入永久标号集合中. 这样的标号算法称为标号设定算法，因为每次迭代可以设定一个节点标号为永久标号. 标号设定算法一般只能用来求解本节中介绍的无圈网络和正费用网络的最短路问题.

对于一般费用网络的最短路问题，一般采用标号修正算法. 它每次迭代时并不一定将任何节点标号从临时标号转变为永久标号，也就是说，在中间的迭代过程中，只是对临时标号进行一次修正，所有节点标号仍然都是临时标号；只有在所有迭代终止时，所有节点标号同时转变为永久标号. 因此，标号设定算法可以看成是标号修正算法的特例，因为在算法终止之前，任何永久标号都可以看成是一种特殊的临时标号. 我们在 4.3 节中介绍标号修正算法.

4.3 一般费用网络：标号修正算法

对于一般费用网络的最短路问题，一般采用标号修正算法. 它的基本思路是：在每次迭代过程中，所有节点的距离标号都是临时标号. 实际上，每个节点的距离标号表示的是在一定限制条件下，从起点到该节点的最短路的路长. 当迭代终止时，限制条件被完全取消，因此所有节点标号同时转变为永久标号. 这一方法实质上是采用逐步逼近的思想求解最短路方程，因此有时也称为逐次逼近法(successive approximation).

4.3.1 Bellman-Ford 算法

我们首先介绍 Bellman-Ford 算法，该算法计算从起点到所有其他节点的最短路. 它是最早提出的一种标号修正算法，由 Ford 于 1956 年提出. 这一算法可以用迭代方程表示如下：

$$\begin{cases} u_1^{(1)}=0, & (4.9)\\ u_j^{(1)}=w_{1j}, \quad i\neq 1, & (4.10)\\ u_j^{(k+1)}=\min\{u_j^{(k)},\min\limits_{i\neq j}\{u_i^{(k)}+w_{ij}\}\}, \quad 1\leqslant k\leqslant n-2,1\leqslant j\leqslant n. & (4.11)\end{cases}$$

通过迭代求解上面的方程，$u_j^{(k)}$ 是第 k 次迭代得到的节点 $j(1\leqslant j\leqslant n)$ 的临时标号，最后得到的 $u_j=u_j^{(n-1)}$ 即为从起点 $s=1$ 到节点 $j(1\leqslant j\leqslant n)$ 的最短路路长（永久标号）. 在算法执行过程中，与 Dijkstra 算法中用 pred 数组记录中间信息类似，还应该记录相应的中间信息，以便最后确定最短路.

下面证明这一算法的正确性. 我们首先证明如下引理.

引理 4.2　在式(4.9)～式(4.11)中，临时标号 $u_j^{(k)}$ 是从起点 $s=1$ 到节点 $j(1\leqslant j\leqslant n)$ 且所经过的弧数不超过 k 条时的最短路路长.

证明　用归纳法证明. 上述结论对临时标号 $u_j^{(1)}$ 显然成立. 假设上述结论对 k 成立，下面考虑 $k+1$ 的情况.

从起点 $s=1$ 到节点 $j(1\leqslant j\leqslant n)$ 且所经过的弧数不超过 $k+1$ 条时的最短路有两种可能：①如果该最短路只含有不超过 k 条弧，则最短路路长为 $u_j^{(k)}$. ②如果该最短路含有 $k+1$ 条弧，设最后一条弧为 (i,j)，则剩下的子路正好含有 k 条弧，子路最短路路长为 $u_i^{(k)}$，故该最短路路长为 $u_i^{(k)}+w_{ij}$. 在这两种情况中取最小值，正好就是 $u_j^{(k+1)}$，因此引理成立. □

从计算过程知道，$u_j^{(k)}$ 随着 k 的增加是非增的，那么是否一定在有限步迭代后收敛呢？对于无负有向圈的网络，最短路中弧的条数不超过 $n-1$ 条. 因此根据以上引理，当 $k\geqslant n-1$ 时一定有 $u_j^{(k)}$ 就是最短路路长 u_j，即这一算法一定在 $n-1$ 步迭代后收敛，所以算法是正确的. 并且，容易看出算法的主要工作量是(4.11)式的循环迭代，对给定的 k 和 j，只需检查节点 j 的所有入弧即可. 所以，对给定的 k，正好需要对网络中的每条弧检查一次. 因此，算法的总复杂度为 $O(mn)$. 这是复杂度相当低的一个强多项式时间算法.

反过来看，如果 $u_j^{(n-1)}$ 尚未收敛，即存在某个节点 j 使得 $u_j^{(n)}<u_j^{(n-1)}$，则说明网络本来就含有负有向圈. 因此本算法也可以用于判断网络是否含有负有向圈.

例 4.6　计算如下网络(图 4.7 (a))中从节点 1 到所有其他节点的最短路.

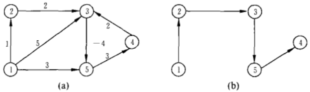


图 4.7　一般费用网络的最短路例子(例 4.6)

计算过程(这里没有详细给出确定最短路的过程，请读者补上)：

$$u_1^{(1)}=0,\quad u_2^{(1)}=1,\quad u_3^{(1)}=5,\quad u_4^{(1)}=\infty,\quad u_5^{(1)}=3.$$
$$u_1^{(2)}=0,\quad u_2^{(2)}=1,\quad u_3^{(2)}=3,\quad u_4^{(2)}=6,\quad u_5^{(2)}=1.$$
$$u_1^{(3)}=0,\quad u_2^{(3)}=1,\quad u_3^{(3)}=3,\quad u_4^{(3)}=4,\quad u_5^{(3)}=-1.$$
$$u_1^{(4)}=0,\quad u_2^{(4)}=1,\quad u_3^{(4)}=3,\quad u_4^{(4)}=2,\quad u_5^{(4)}=-1.$$

最后得到的最短路树形图如图 4.7(b)所示. □

4.3.2 一般的标号修正算法

根据定理 4.1，对于只含正圈的连通有向网络，最短路的长度可以由最短路方程唯一确定. 但由于根据最短路方程(4.7)求解 u_j 时必须知道所有其他节点 i 上的 u_i 值，因此一般情况下直接求解最短路方程是相当困难的. 标号修正算法实质上是采用逐步逼近的思想求解最短路方程，也就是要使最后获得的距离标号满足最短路方程. 如果当前的距离标号不满足最短路方程，我们就对距离标号进行修改. 因此，一般的标号修正算法可以如下描述：

一般的标号修正算法

STEP0 令距离标号 $u_s=0$，前趋标号 pred$(s)=0$；对所有其他节点 j 令 u_j 为无穷大.

STEP1 如果对所有的弧(i,j)有 $u_j \leqslant u_i + w_{ij}$，则结束，$u_j$ 就是从起点 s 到节点 j 的最短路长，最短路可以通过前趋标号(pred)获得. 否则进行下一步.

STEP2 找到一条满足 $u_j > u_i + w_{ij}$ 的弧(i,j)，令 $u_j = u_i + w_{ij}$，pred$(j)=i$. 转 STEP1.

我们总是假设网络中没有负圈，且所有的权为整数(为有理数时也一样). 由于每次迭代使得一个节点的距离标号至少减少 1，因此对每个节点的距离标号的修正次数不超过 $2nC$ 次(这里 C 为网络 G 中最大弧(按绝对值)的费用值(权)). 因此，总迭代次数不会超过 $2n^2C$ 次，算法一定在有限步内结束. 然而，这种复杂度的算法只是伪多项式时间算法，而不是真正的多项式时间算法. 此外，上面的一般算法在 STEP2 中没有给出任何一种具体方法，以便找到一条满足 $u_j > u_i + w_{ij}$ 的弧(i,j). 如果每次迭代都对所有弧进行检查和判断，则非常花费时间，需要 m 次操作，即算法的总复杂度为 $O(mn^2C)$.

一种自然的想法是：能否将不满足 $u_j \leqslant u_i + w_{ij}$ 的弧(i,j)记录下来，以便方便地对其端点 j 的距离标号进行修正呢？假设用链表 LIST 记录这些弧，那么，当我们每次从中取出一条弧(i,j)并对 u_j 进行修正(减少)后，则从节点 j 出发的所有弧都有可能需要加入到 LIST 中. 因此，实际上我们记录下一步需要修改标号的节点比记录弧更方便. 这样得到的算法可以称为改进的标号修正算法，现描述如下：

改进的标号修正算法

STEP0 令 LIST$=\{s\}$，距离标号 $u_s=0$，前趋标号 pred$(s)=0$；对所有其他节点 j 令 u_j 为无穷大.

STEP1 如果 LIST$=\varnothing$，则结束，u_j 就是从起点 s 到节点 j 的最短路长，最短路可以通过前趋标号(pred)回溯获得. 否则进行下一步.

STEP2 从 LIST 中删去一个节点 i，对从 i 出发的每条弧(i,j)判断是否满足 $u_j > u_j + w_{ij}$.

如果是，则令 $u_j=u_i+w_{ij}$，$\text{pred}(j)=i$，并令 LIST＝LIST$\cup\{j\}$. 当从 i 出发的所有弧都检查完以后，转 STEP1.

前面已经分析过，对每个节点的距离标号的修正次数不超过 $2nC$ 次. 因此，这一算法的总复杂度为 $\sum(2nC)d^+(i)=O(mnC)$，这里 $d^+(i)$ 表示节点 i 的出度. 然而，这种复杂度的算法仍然不是真正的多项式时间算法（只是伪多项式时间算法）.

但是，可以通过非常简单的实现技巧使得算法成为多项式时间算法. 例如，在一般的标号修正算法中，可以首先对所有弧给定一个顺序，然后依次检查每条弧 (i,j) 并且在必要时对 u_j 进行修正（减少）；当所有弧均被检查一遍以后，再从第 1 条弧开始下一遍检查. 从计算过程知道，经过 k 遍检查以后，节点 j 所获得的距离标号 u_j 就是 Bellman-Ford 算法中的 $u_j^{(k)}$，即从起点 $s=1$ 到节点 $j(1\leqslant j\leqslant n)$ 且所经过的弧数不超过 k 条时的最短路路长. 也就是说，Bellman-Ford 算法正是这样一种特殊的实现方法. 因此根据前面讨论，这一算法一定在对所有弧检查 $n-1$ 遍后收敛. 因此，算法的总复杂度为 $O(mn)$. 这是复杂度相当低的一个强多项式时间算法.

仔细分析可以知道，标号设定算法是一般标号修正算法的特例. 实际上，我们前面介绍的所有最短路算法都是一般标号修正算法的特例.

4.3.3　Floyd-Warshall 算法

下面介绍计算网络中所有节点之间的最短路的算法. 自然，可以通过 n 次调用前面的算法来完成这一任务，但复杂度可能太高. 如 n 次调用 Bellman-Ford 算法的复杂度为 $O(n^2m)$. 下面介绍 Floyd-Warshall 算法（1962 年提出），该算法计算网络中所有节点之间的最短路，复杂度为 $O(n^3)$. 这一算法可以用迭代方程表示如下：

$$\begin{cases} u_{ii}^{(1)}=0, & \\ u_{ij}^{(1)}=w_{ij}, & i\neq j, \\ u_{ij}^{(k+1)}=\min\{u_{ij}^{(k)},u_{ik}^{(k)}+u_{kj}^{(k)}\}, & i,j,k=1,2,\cdots,n. \end{cases} \tag{4.12}$$

这种算法可以看成是标号修正算法的一种推广. 通过迭代求解上面的方程，$u_{ij}^{(k)}$ 是第 k 次迭代得到的节点 i,j 的临时标号，最后得到的 $u_{ij}=u_{ij}^{(n+1)}$ 即为从节点 i 到节点 j 的最短路路长（永久标号）.

下面证明这一算法的正确性. 我们首先证明如下引理：

引理 4.3　在迭代方程(4.12)中，临时标号 $u_{ij}^{(k)}$ 是不通过 $k,k+1,\cdots,n$ 节点（i,j 除外）时从节点 i 到节点 j 的最短路路长.

证明　用归纳法证明. 上述结论对临时标号 $u_{ij}^{(1)}$ 显然成立. 假设上述结论对 k 成立，下面考虑 $k+1$ 的情况.

从起点 i 到节点 j 且不经过 $k+1,k+2,\cdots,n$ 的最短路有两种可能：①如果该最短路不经过 k 节点，则最短路路长为 $u_{ij}^{(k)}$. ②如果该最短路经过 k 节点，则该最短路路长为由 k

节点分开的两条子路的最短路路长之和，即 $u_{ik}^{(k)}+u_{kj}^{(k)}$. 在这两种情况中取最小值，正好就是 $u_{ij}^{(k+1)}$，因此引理成立. □

根据以上引理，当 $k\geqslant n+1$ 时一定有 $u_{ij}^{(k)}$ 就是最短路路长 u_{ij}，即这一算法一定在 n 步迭代后收敛，所以算法是正确的. 并且，容易看出算法的主要工作量是一个 3 重循环，因此复杂度为 $O(n^3)$.

反过来看，如果 $u_{ij}^{(n+1)}$ 尚未收敛，即存在某两个节点 i，j 使得 $u_{ij}^{(n+2)}<u_{ij}^{(n+1)}$，则说明网络本来就含有负有向圈. 其实，如果网络不含有负有向圈，在迭代过程中对任意 k 应当都有 $u_{ii}^{(k)}=0$. 如果在某次迭代 k 发现某个节点 i 使得 $u_{ii}^{(k)}<0$，则说明网络含有负有向圈. 因此本算法也可以用于判断网络是否含有负有向圈.

在算法的具体实施过程中，与 Dijkstra 算法中用 pred 数组记录中间信息类似，还应该记录相应的中间信息，以便最后确定最短路. 由于要记录所有节点之间最短路的信息，所以这里要用一个二维数组 $\boldsymbol{P}=(p_{ij})_{n\times n}$. 具体来说，可以用 p_{ij} 表示从节点 i 到 j 的当前最短路中第 1 条弧的头节点. 最后，可以依据二维数组 $\boldsymbol{P}$，采用"正向追踪"的方式得到最短路. 显然，开始时 $p_{ij}^{(1)}=j$. 算法可以具体描述如下：

Floyd-Warshall 算法

STEP0　$k=0$. 对于所有节点 i 和 j，令 $p_{ij}^{(1)}=j$，$u_{ii}^{(1)}=0$（可以认为 $w_{ii}=0$），$u_{ij}^{(1)}=w_{ij}$ $(i\neq j)$（若节点 i 和 j 之间没有弧，认为 $w_{ij}=\infty$）.

STEP1　$k=k+1$. 对于所有节点 i 和 j，若 $u_{ij}^{(k)}\leqslant u_{ik}^{(k)}+u_{kj}^{(k)}$，令 $p_{ij}^{(k+1)}=p_{ij}^{(k)}$，$u_{ij}^{(k+1)}=u_{ij}^{(k)}$；否则令 $p_{ij}^{(k+1)}=p_{i,k}^{(k)}$，$u_{ij}^{(k+1)}=u_{ik}^{(k)}+u_{kj}^{(k)}$.

STEP2　如果 $k=n$，结束；否则转 STEP1.

例 4.7　计算如下网络（图 4.8）中所有节点之间的最短路.

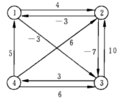


图 4.8　所有节点之间最短路的例子（例 4.7）

记最短路长矩阵为 $\boldsymbol{U}$，最短路矩阵为 $\boldsymbol{P}$，都按照节点 1-2-3-4 的顺序存放有关信息.

初始值为

$$\boldsymbol{U}^{(1)}=\begin{bmatrix}0&4&-3&\infty\\-3&0&-7&\infty\\\infty&10&0&3\\5&6&6&0\end{bmatrix},\quad \boldsymbol{P}^{(1)}=\begin{bmatrix}1&2&3&4\\1&2&3&4\\1&2&3&4\\1&2&3&4\end{bmatrix},$$

第 1 次迭代后得到（只有 p_{43} 和 u_{43} 发生了变化）：

$$U^{(2)}=\begin{bmatrix}0&4&-3&\infty\\-3&0&-7&\infty\\\infty&10&0&3\\5&6&2&0\end{bmatrix},\quad P^{(2)}=\begin{bmatrix}1&2&3&4\\1&2&3&4\\1&2&3&4\\1&2&1&4\end{bmatrix},$$

第2次迭代后得到(只有 p_{31},p_{41},p_{43} 和 u_{31},u_{41},u_{43} 发生了变化):

$$U^{(3)}=\begin{bmatrix}0&4&-3&\infty\\-3&0&-7&\infty\\7&10&0&3\\3&6&-1&0\end{bmatrix},\quad P^{(3)}=\begin{bmatrix}1&2&3&4\\1&2&3&4\\2&2&3&4\\2&2&2&4\end{bmatrix},$$

第3次迭代后得到(只有 p_{14},p_{24} 和 u_{14},u_{24} 发生了变化):

$$U^{(4)}=\begin{bmatrix}0&4&-3&0\\-3&0&-7&-4\\7&10&0&3\\3&6&-1&0\end{bmatrix},\quad P^{(4)}=\begin{bmatrix}1&2&3&3\\1&2&3&3\\2&2&3&4\\2&2&2&4\end{bmatrix},$$

第4次迭代后得到(只有 p_{31},p_{32} 和 u_{31},u_{32} 发生了变化):

$$U^{(5)}=\begin{bmatrix}0&4&-3&0\\-3&0&-7&-4\\6&9&0&3\\3&6&-1&0\end{bmatrix},\quad P^{(5)}=\begin{bmatrix}1&2&3&3\\1&2&3&3\\4&4&3&4\\2&2&2&4\end{bmatrix}.$$

最后得到的最短路路长可以直接从 $U^{(5)}$ 得到. 根据对应的 $P^{(5)}$，最后得到的最短路为：

终点		1	2	3	4
起点	1		(1,2)	(1,3)	(1,3)，(3,4)
	2	(2,1)		(2,3)	(2,3)，(3,4)
	3	(3,4),(4,2),(2,1)	(3,4),(4,2)		(3,4)
	4	(4,2),(2,1)	(4,2)	(4,2),(2,3)	

练 习 题

1. 证明下列说法正确或给出反例说明其不成立.

(1) 当网络不含有向圈时，其最短路方程(Bellman 方程)的解唯一.

(2) 如果有向网络中所有弧上的权都不相同，则最短路树唯一.

(3) 如果有向网络中所有弧上的权都是正数，则 Dijkstra 算法求得的最短路是所有最短路中包含弧数最少的最短路.

(4) 如果有向网络中所有弧上的权都是正数，则将所有弧的方向去掉以后(即变成为无向网络)，最短路的路长不会发生变化.

(5) 如果将有向网络中所有弧上的权同时增加 k 个单位($k>0$)，则最短路的路长增加 k 个单位的一个整数倍数.

(6) 如果将有向网络中所有弧上的权同时减少 k 个单位($k>0$)，则最短路的路长减少 k 个单位的一个整数倍数.

(7) 如果某无向网络中节点 1 到其他各节点均有唯一的最短路，则连接节点 1 到其他各节点的最短路在去掉重复部分后，恰好构成该图的最小树.

(8) 不存在这样的有向网络，它的最短路树包含网络中的最大弧，但不包含最小弧.

2. 试将背包问题(参见例 1.7)转化为最短路问题.

3. 已知某有向网络的弧长矩阵如下(空白元素表示对应的节点之间没有连接弧)，计算网络中从起点(节点 1)到终点(节点 7)的最短路和路长，以及最长路和路长.

$$\begin{bmatrix} 0 & 4 & 3 & 5 & & 9 & 6 \\ & 0 & -1 & & 6 & 4 & 0 \\ & & 0 & 2 & -10 & 3 & -9 \\ & & & 0 & 8 & 2 & 8 \\ & & & & 0 & 1 & 6 \\ & & & & & 0 & 3 \\ & & & & & & 0 \end{bmatrix}$$

4. 计算如图 4.9 所示无圈网络中从节点 1 到所有其他节点的最短路和路长.

5. 用 Dijkstra 算法计算如图 4.10 所示网络中从节点 1 到所有其他节点的最短路和路长.

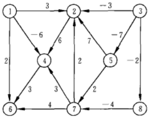


图 4.9　第 4 题图

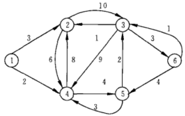


图 4.10　第 5 题图

6. 已知某有向网络的弧长矩阵如下(空白元素表示对应的节点之间没有连接弧)，用 Bellman-Ford 算法计算网络中从起点(节点 1)到所有其他节点的最短路和路长.

$$\begin{bmatrix} 0 & & 4 & 10 & 3 & & \\ & 0 & -1 & -3 & 2 & 11 & \\ & 9 & 0 & 8 & 3 & 2 & 1 \\ & 4 & & 0 & 8 & 6 & 3 \\ & & 1 & 2 & 0 & 3 & -1 \\ & -1 & -1 & 3 & 2 & 0 & \\ & 4 & 3 & & & 2 & 0 \end{bmatrix}$$

7. 对第5、6题，用Floyd-Warshall算法计算所有节点之间的最短路路长.

8. 在Floyd-Warshall算法中，如何在求得所有最短路路长的同时，得到所有最短路?

9. 仿照将Bellman-Ford算法推广到一般标号修正算法的思路，将Floyd-Warshall算法推广到可以计算所有节点间最短路的一般标号修正算法，并分析计算复杂度.

10. 6个村庄之间的道路和距离如图4.11所示.现拟合建一所小学，已知A村有小学生50人，B村40人，C村60人，D村20人，E村70人，F村90人.小学应建在哪一个村庄，可使学生上学最方便(即走的总路程最短)?

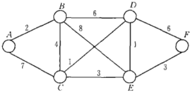


图4.11　第10题图

11. 在有向网络中，对于给定的起点和终点，将所有有向路按照路长非降的顺序排列，称处于第k个位置的有向路为第k最短路.试设计一个算法，计算有向网络的第2最短路，并分析算法的计算复杂度.

12. 在有向网络中，对于给定的起点和终点，如果去掉一条弧后使得最短路的路长严格增加，则称该弧为"活跃弧". 去掉后使得最短路路长增加最多的一条弧称为"最活跃弧".试设计一个算法，计算有向网络的"最活跃弧"，并分析算法的计算复杂度.

13. 最短路的敏感性分析:设计一个算法，对有向网络的每条弧上的权计算一个区间，使得当权在该区间内变化时，最短路树不变.分析算法的计算复杂度.

14. 考虑具有两个权函数的有向网络:每条弧(i,j)上赋予权w_{ij}和c_{ij} ($c_{ij} \geqslant 0$)，并设对于该网络的任意有向圈Q，有$\sum_{(i,j)\in Q} c_{ij} > 0$.试设计一个算法，对于给定的数值$a$，判断该网络中是否存在有向圈$Q^*$，使得$\sum_{(i,j)\in Q^*} w_{ij} \Big/ \sum_{(i,j)\in Q^*} c_{ij} > a$?并分析算法的计算复杂度.

15. 考虑具有两个权函数的有向网络:每条弧(i,j)上赋予权w_{ij}和c_{ij} ($c_{ij} > 0$). 如下的问题称为有约束的最短路问题:给定数值a，在按照权c_{ij}计算的路长不超过a的路中，计算该网络中某节点到其他所有节点按照权w_{ij}的最短路.试设计一个算法求解此问题，并分析算法的计算复杂度.

第 5 章　最大流问题

本章讨论的最大流问题和下一章将要讨论的最小费用流问题，都是以网络中的流为研究对象的. 所谓网络中的流，其意义类似于在网络中将一些"物质"从一个节点沿着弧发送到另一个节点. 我们首先看下面的例子：

假设从城市 P 到城市 Q 的公路网如图 5.1 所示. 假设每段公路上每天可以通过的汽车的数量有一个上限(弧上用一个数字表示，单位为万辆)，那么经过该公路网，每天最多可以有多少辆汽车从城市 P 开到城市 Q? 由于考虑的是汽车从城市 P 出发到城市 Q 终止的情况，而不考虑汽车从城市 Q 返回城市 P 的问题，所以图 5.1 中没有画出从城市 Q 出发或终止于城市 P 的弧. 此外，从图中可以看出，城市 A，B，C，D 之间的有些道路是双向的(即图中 AC 之间和 BD 之间)，而另外一些则是单向的.

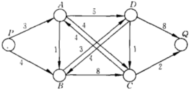


图 5.1　最大流问题的例子(公路网络)

可以看出，这一问题考虑的是每天从 P 到 Q 的最大车流量，这样的问题在网络优化中称为最大流问题. 本章我们首先介绍流网络和网络中的流的基本概念，然后讨论最大流问题的一些基本性质和主要算法.

5.1　最大流问题的数学描述

5.1.1　网络中的流

定义 5.1　在以 V 为节点集，A 为弧集的有向图 $G=(V, A)$ 上定义如下的权函数：

(1) $L: A\to\mathbb{R}$ 为弧上的权函数，弧 $(i, j)\in A$ 对应的权 $L(i, j)$ 记为 l_{ij}，称为弧 (i, j) 的容量下界(lower bound)；

(2) $U: A\to\mathbb{R}$ 为弧上的权函数，弧 $(i, j)\in A$ 对应的权 $U(i, j)$ 记为 u_{ij}，称为弧 (i, j) 的容量上界，或直接称为容量(capacity)；

(3) $D: V\to\mathbb{R}$ 为节点上的权函数，节点 $i\in V$ 对应的权 $D(i)$ 记为 d_i，称为节点 i 的供需量(supply/demand).

此时所构成的网络称为流网络，可以记为 $N=(V, A, L, U, D)$.

由于本书只讨论 V，A 为有限集合的情况，所以对于弧上的权函数 L，U 和节点上的权函数 D，可以直接用所有弧上对应的权组成的有限维向量表示，因此 L，U，D 有时直接

称为权向量，或简称权. 由于给定有向图 $G=(V, A)$后，我们总是可以在它的弧集合和节点集合上定义各种权函数，所以流网络一般也直接简称为网络. 请读者注意，第1章中我们用 $d(i)$表示节点 i 的度数，而这里我们用符号 d_i 表示节点 i 的供需量，请不要产生误解.

在流网络中，弧(i, j)的容量下界 l_{ij} 和容量上界 u_{ij} 表示的物理意义分别是：通过该弧发送某种"物质"时，必须发送的最小数量为 l_{ij}，而发送的最大数量为 u_{ij}. 节点 $i\in V$ 对应的供需量 d_i 则表示该节点从网络外部获得的"物质"数量($d_i>0$ 时)，或从该节点发送到网络外部的"物质"数量($d_i<0$ 时). 下面我们给出严格定义.

定义 5.2 对于流网络 $N=(V, A, L, U, D)$，其上的一个流(flow)x 是指从 N 的弧集 A 到 $\mathbb{R}$ 的一个函数，即对每条弧$(i, j)\in A$ 赋予一个实数 x_{ij}(称为弧(i, j)的流量). 如果流 x 满足

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = d_i, \quad \forall i \in V, \tag{5.1}$$

$$l_{ij} \leqslant x_{ij} \leqslant u_{ij}, \quad \forall (i, j) \in A, \tag{5.2}$$

则称 x 为可行流(feasible flow). 至少存在一个可行流的流网络称为可行网络(feasible network). 约束条件(5.1)称为流量守恒条件(也称流量平衡条件)，约束条件(5.2)称为容量约束. 如果流 x 只满足约束条件(5.2)，但不一定满足约束条件(5.1)，则称 x 为伪流(pseudoflow)，或容量可行流.

可见，当 $d_i>0$ 时，表示有 d_i 个单位的流量从网络外部流入该节点，因此节点 i 称为供应点(supply node)或源(source)，有时也形象地称为起始点或发货点等；当 $d_i<0$ 时，表示有 d_i 个单位的流量从该节点流失到网络外部(或说被该节点吸收)，因此节点 i 称为需求点(demand node)或汇(sink)，有时也形象地称为终止点或吸收点等；当 $d_i=0$ 时，节点 i 称为转运点(transshipment node)或平衡点、中间点等. 此外，根据式(5.1)可知，对于可行网络，必有

$$\sum_{i\in V} d_i = 0. \tag{5.3}$$

也就是说，所有节点上的供需量之和为 0 是网络中存在可行流的必要条件.

例 5.1 在一个有向网络中，令所有弧上的容量下界为 0，容量(上界)为 1，并令图中某个节点 s 的供需量为 1，某个节点 t 的供需量为 -1，则从 s 到 t 的一条有向路正好对应于网络中的一个可行流 x(当 $x_{ij}=1$ 时，表示弧(i, j)位于 s-t 路上，当 $x_{ij}=0$ 时，表示弧(i, j)不在 s-t 路上). 实际上，在第 4 章的最短路问题建模(4.1)～(4.3)中，我们正是用这样的方式来描述一条路的. 值得指出的是，网络中的任意一个可行流却不一定对应于一条有向路. □

一般来说，我们总是可以把 $L\neq 0$ 的流网络转化为 $L=0$ 的流网络进行研究(请读者思考这是为什么，并尝试给出一个证明). 所以，除非特别说明，以后我们总是假设 $L=0$(即所有弧(i, j)的容量下界 $l_{ij}=0$)，并将 $L=0$ 时的流网络简记为 $N=(V, A, U, D)$. 此时，相应的容量约束(5.2)为

$$0 \leqslant x_{ij} \leqslant u_{ij}, \quad \forall (i, j) \in A. \tag{5.2a}$$

定义 5.3　在流网络 $N=(V, A, U, D)$中，对于流 x，如果

$$x_{ij}=0, \quad \forall (i, j) \in A,$$

则称 x 为零流，否则为非零流. 如果某条弧(i, j)上的流量等于其容量$(x_{ij}=u_{ij})$，则称该弧为饱和弧(saturated arc)；如果某条弧(i, j)上的流量为 $0(x_{ij}=0)$，则称该弧为空弧(void arc).

为了以后讨论的需要，我们下面介绍所谓路流、圈流的概念，然后介绍流的分解定理.

定义 5.4　对于给定流网络 $N=(V, A, U, D)$中的流 $x: A \to \mathbb{R}$，如果 N 中存在一条有向路 P，使得

$$0 \leqslant v = x_{ij} \leqslant u_{ij}, \quad \forall (i, j) \in P,$$
$$x_{ij} = 0, \quad \forall (i, j) \in A \backslash P,$$

则称 x 为路流(path flow)，v 称为该路流的流值(流量). $v=0$ 时，称该路流为零路流，否则称为非零路流.

同样，如果 N 中存在一个有向圈 W，使得

$$0 \leqslant v = x_{ij} \leqslant u_{ij}, \quad \forall (i, j) \in W,$$
$$x_{ij} = 0, \quad \forall (i, j) \in A \backslash W,$$

则称 x 为圈流(cycle flow)，v 称为该圈流的流值(流量). $v=0$ 时，称该圈流为零圈流，否则称为非零圈流.

定理 5.1(流的分解定理，flow decomposition theorem)　在流网络 $N=(V, A, U, D)$中，任何一个可行流一定可以表示为若干个路流和若干个圈流之和，且这些路流和圈流满足下列性质：

(1) 非零路流对应的有向路从一个源点指向一个汇点；

(2) 至多有 $m+n$ 个路流和圈流为非零流，且其中至多有 m 个圈流为非零流.

证明　我们给出一个构造性证明. 记可行流为 x，设 i_0 是网络中的一个源点，则存在弧$(i_0, i_1) \in A$ 使得 $x_{i_0 i_1}>0$. 如果 i_1 是网络中的一个汇点，则找到了一条从一个源点指向一个汇点的有向路. 否则，从 i_1 出发，重复上述过程，直到找到一个汇点或再次遇到前面已经经过的某个节点时为止，即直到下列两种情况之一出现为止：

① 找到一条从一个源点 i_0 指向一个汇点 i_k 的有向路 P. 此时，定义

$$v(P)=\min\{d_{i_0}, -d_{i_k}, \min\{x_{ij} \mid (i, j) \in P\}\},$$

则我们构造了一个流值为 $v(P)$的非零路流. 此时，在网络中重新定义

$$d_{i_0}=d_{i_0}-v(P), \quad d_{i_k}=d_{i_k}+v(P),$$
$$x_{ij}=x_{ij}-v(P), \quad \forall (i, j) \in P.$$

② 找到一个有向圈 W. 此时，定义

$$v(W)=\min\{x_{ij} \mid (i, j) \in W\},$$

则我们构造了一个流值为 $v(W)$的非零圈流. 此时，在网络中重新定义

$$x_{ij}=x_{ij}-v(W),\quad \forall (i,j)\in W.$$

对重新定义的网络，重复上述过程，直到所有节点变成为转运点(平衡点). 然后，对重新定义的网络，我们找到一个至少有一条出弧上的流量为正的节点，继续重复上述过程，这时只有情形②会出现，即一定获得一个非零圈流. 重复上述过程，直到重新定义的流 x 成为零流为止.

从这一构造过程可以知道，上述过程中所找到的所有路流和圈流的和，等于原始可行流 x. 进一步，当我们找到一个路流时，总是通过重新定义使得某个节点的供需量从非零成为零，或者使得某条弧的流量从非零成为零. 当我们找到一个圈流时，我们总是通过重新定义使得某条弧的流量从非零成为零.

因此，上述过程找到路流和圈流的次数之和不会多于 $m+n$ 次，且其中找到圈流的次数不会多于 m 次. □

当网络中无源无汇，即所有节点都是转运点，亦即所有节点上的供需量都是 0 时，该网络中的一个可行流称为可行循环流. 对可行循环流，我们可以得到如下推论：

推论 5.1 在一个无源无汇的流网络中，一个可行循环流一定可以表示为至多 m 个非零圈流之和. □

5.1.2 最大流问题

考虑如下流网络 $N=(V,A,U,D)$：节点 s 为网络中唯一的源点，t 为唯一的汇点，而其他节点为转运点. 如果网络中存在可行流 x，此时称流 x 的流量(或流值，flow value)为 d_s(根据式(5.3)，它自然也等于 $-d_t$)，通常记为 v 或 $v(x)$，即 $v=v(x)=d_s=-d_t$.

对这种单源单汇的网络，如果我们并不给定 d_s 和 d_t(即流量不给定)，则网络一般记为 $N=(s,t,V,A,U)$. 此时，也常常把容量可行(即满足式(5.2))且转运点流量守恒的流称为 s-t 可行流，有时为了方便也称为可行流. 最大流问题(maximum flow problem)就是在 $N=(s,t,V,A,U)$ 中找到流值最大的 s-t 可行流(即最大流). 我们后面将会看到，最大流问题的许多算法也可以用来求解流量给定的网络中的可行流. 也就是说，当我们解决了最大流问题以后，对于在流量给定的网络中寻找可行流的问题，通常也就可以解决了.

因此，用数学规划的方法，最大流问题可以形式地描述如下：

$$\max v$$

$$\text{s.t.}\quad \sum_{j:(i,j)\in A} x_{ij}-\sum_{j:(j,i)\in A} x_{ji}=\begin{cases} v, & i=s \\ -v, & i=t, \\ 0, & i\neq s,t, \end{cases} \tag{5.4}$$

$$0\leqslant x_{ij}\leqslant u_{ij},\quad \forall (i,j)\in A. \tag{5.2a}$$

定理 5.2(整流定理) 最大流问题所对应的约束矩阵是全幺模矩阵. 若所有弧容量均为正整数，则问题的最优解为整数解.

证明 设节点的编号为 1(即 s)，2，…，$n-1$，n (即 t)，弧表按节点的字典序排列. 记

网络的关联矩阵为 $\boldsymbol{B}$，并记

$$\boldsymbol{x}=\begin{pmatrix} x_{11} \\ x_{12} \\ \vdots \\ x_{ij} \\ \vdots \end{pmatrix}, \quad \boldsymbol{J}=\begin{pmatrix} -1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}, \quad \boldsymbol{U}=\begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{ij} \\ \vdots \end{pmatrix}$$

则约束条件(5.4)为

$$\boldsymbol{Bx}+v\boldsymbol{J}=\boldsymbol{0},$$

即

$$(\boldsymbol{B} \quad \boldsymbol{J})\begin{pmatrix} \boldsymbol{x} \\ v \end{pmatrix}=\boldsymbol{0}.$$

引入松弛变量 $\boldsymbol{y}$，则约束条件(5.2a)为

$$\boldsymbol{x}+\boldsymbol{y}=\boldsymbol{U}.$$

因此，最大流问题可以用矩阵形式表示为

$$\begin{aligned} &\max\ v \\ &\text{s. t.}\ \begin{pmatrix} \boldsymbol{B} & \boldsymbol{J} & \boldsymbol{0} \\ \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix}\begin{pmatrix} \boldsymbol{x} \\ v \\ \boldsymbol{y} \end{pmatrix}=\begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{U} \end{pmatrix}, \\ &v\geqslant 0, \quad \boldsymbol{x},\ \boldsymbol{y}\geqslant \boldsymbol{0}, \end{aligned}$$

其中 $\boldsymbol{I}$ 和 $\boldsymbol{0}$ 分别表示相应维数的单位矩阵和零矩阵.

因为关联矩阵 $\boldsymbol{B}$ 为全幺模矩阵，所以上述线性规划问题的约束矩阵为全幺模矩阵. 因此，当所有弧容量均为正整数时，问题的最优解为整数解. □

5.1.3 增广路定理

下面再介绍与最大流问题有关的一些基本概念和基本定理.

定义 5.5 设$[S, T]$是网络 $N=(s, t, V, A, U)$中给定的一个割，且 $s\in S$，$t\in T$，则称割$[S, T]$为 s-t 割. s-t 割$[S, T]$中的弧$(i, j)(i\in S, j\in T)$称为割的前向弧，弧$(i, j)(j\in S, i\in T)$称为割的反向弧. s-t 割$[S, T]$的容量定义为前向弧的容量之和，记为

$$U(S, T) = \sum_{\substack{(i, j)\in A, \\ i\in S, j\in T}} u_{ij}. \tag{5.5}$$

一个网络中容量最小的 s-t 割称为最小割.

引理 5.1 任意一个 s-t 可行流的流值不超过任意一个 s-t 割的容量.

证明 设 $\boldsymbol{x}$ 为一个 s-t 可行流，$[S, T]$为一个 s-t 割，则由式(5.4)有

$$v(\boldsymbol{x}) = \sum_{i\in S}\Big(\sum_{j\in V}x_{ij} - \sum_{j\in V}x_{ji}\Big)$$

$$
\begin{aligned}
&= \sum_{i\in S}\sum_{j\in S}(x_{ij}-x_{ji}) + \sum_{i\in S}\sum_{j\in T}(x_{ij}-x_{ji}) \\
&= \sum_{i\in S}\sum_{j\in T}(x_{ij}-x_{ji}) \\
&\leqslant \sum_{i\in S}\sum_{j\in T}x_{ij} \quad (\text{因为 } x_{ji}\geqslant 0) \\
&\leqslant \sum_{i\in S}\sum_{j\in T}u_{ij} \quad (\text{因为 } x_{ij}\leqslant u_{ij}) \\
&= U(S,T).
\end{aligned}
$$

因此引理成立. □

从这一证明过程同时可以看出，任意一个 s-t 可行流的流值等于任意一个 s-t 割的所有前向弧的流量之和减去所有后向弧的流量之和. 进一步，如果存在一个可行流的流值等于一个 s-t 割的容量，则该可行流为最大流，该 s-t 割为最小割.

定义 5.6　设 $\boldsymbol{x}$ 是流网络 $N=(s,t,V,A,U)$中给定的可行流，P 是一条 s-t 路，则 P 中满足下列两个条件之一的弧(i,j)称为增广弧(augmenting arc)：

(1) 弧(i,j)是 P 的前向弧(即$(i,j)\in P^+$)且为不饱和弧(即 $x_{ij}<u_{ij}$)；或

(2) 弧(i,j)是 P 的反向弧(即$(i,j)\in P^-$)且为非空弧(即$x_{ij}>0$).

如果 P 中的弧都是增广弧，则称 P 为关于流 $\boldsymbol{x}$ 的增广路，简称增广路(augmenting path).

引理 5.2　如果对于一个可行流存在增广路，则该可行流不是最大流.

证明　设 P 为关于可行流 $\boldsymbol{x}$ 的一条增广路，令

$$\Delta=\min\left\{\min_{(i,j)\in P^+}\{u_{ij}-x_{ij}\},\ \min_{(i,j)\in P^-}\{x_{ij}\}\right\},$$

则 $\Delta>0$. 于是，可以构造一个新的可行流如下(这一过程称为流的增广)：

$$x'_{ij}=\begin{cases} x_{ij}+\Delta, & (i,j)\in P^+, \\ x_{ij}-\Delta, & (i,j)\in P^-, \\ x_{ij}, & (i,j)\notin P. \end{cases}$$

则 $\boldsymbol{x}'$也是可行流，且 $v(\boldsymbol{x}')=v(\boldsymbol{x})+\Delta>v(\boldsymbol{x})$. 故 $\boldsymbol{x}$ 不是最大流. □

上面证明实际上是一个构造性证明，从可行流 $\boldsymbol{x}$ 和增广路 P 构造可行流 $\boldsymbol{x}'$的过程称为对 $\boldsymbol{x}$ 关于 P 的增广(augmentation)，或说 $\boldsymbol{x}'$是 $\boldsymbol{x}$(关于 P)的增广.

定理 5.3(增广路定理)　一个可行流为最大流的充要条件是不存在增广路.

证明　必要性可以由前面的引理直接得到. 下面证明充分性.

假设流网络 $N=(s,t,V,A,U)$中不存在关于可行流 $\boldsymbol{x}$ 的增广路，记网络中从 s 出发沿增广弧可以到达的节点集合为 S，令 $T=V\backslash S$，则 $s\in S$，$t\in T$，即$[S,T]$为 s-t 割. 并且，对于 A 中的任意弧(i,j)，如果它是该 s-t 割的前向弧，则 $x_{ij}=u_{ij}$；如果它是该 s-t 割的后向弧，则 $x_{ij}=0$.

于是利用前面引理 5.1 证明中的中间结果有

$$
\begin{aligned}
v(\boldsymbol{x}) &= \sum_{i\in S}\sum_{j\in T}(x_{ij} - x_{ji}) \\
&= \sum_{i\in S}\sum_{j\in T} u_{ij} \\
&= U(S,\ T).
\end{aligned}
$$

因此，$[S,\ T]$为最小割，$\boldsymbol{x}$ 为最大流. □

上面的证明过程实际上说明了如果网络中存在最大流，则最大流的流值等于最小割的容量. 本章后面介绍的算法可以说明最大流一定存在并且实际上找到最大流，因此可以得到如下的最大流最小割定理.

定理 5.4(最大流最小割定理) 最大流的流值等于最小割的容量. □

5.2 增广路算法

5.2.1 Ford-Fulkerson 标号算法

从 5.1 节可以知道，最大流问题可以用线性规划来描述，因此可以用一般的线性规划算法来求解. 但考虑到问题的特殊性，我们可以有更有效的算法.

根据增广路定理，为了得到最大流，我们可以从任何一个可行流开始，沿增广路对流进行增广，直到网络中不存在增广路为止. 这样的算法称为增广路算法. 问题的关键在于如何有效地找到增广路，并保证算法在有限次增广后一定终止. 本节首先介绍 Ford 和 Fulkerson 于 1956 年提出的标号算法，该算法的思想是通过一个标号过程来寻找网络中的增广路. 对每个节点 j，其标号包括两部分信息(pred(j)，maxf(j))：该节点在可能的增广路中的前一个节点 pred(j)，以及沿该可能的增广路到该节点为止可以增广的最大流量 maxf(j).

算法的具体步骤如下：

Ford-Fulkerson 标号算法(求网络 $N=(s,\ t,\ V,\ A,\ U)$中的最大流 $\boldsymbol{x}$)

STEP0 置初始可行流 $\boldsymbol{x}$(如零流)；对节点 t 标号，即令 maxf(t)＝任意正值(如 1).

STEP1 若 maxf(t)＞0，继续下一步；否则停止，已经得到最大流，结束.

STEP2 取消所有节点 $j\in V$ 的标号，即令 maxf(j)＝0，pred(j)＝0；令 LIST＝$\{s\}$；对节点 s 标号，即令 maxf(s)＝充分大的正值.

STEP3 如果 LIST≠∅且 maxf(t)＝0，继续下一步；否则：

STEP3.1 如果 t 已经有标号(即 maxf(t)＞0)，则找到了一条增广路，沿该增广路对流 $\boldsymbol{x}$ 进行增广(增广的流量为 maxf(t)，增广路可以根据 pred 回溯方便地得到)，转 STEP1.

STEP3.2 如果 t 没有标号(即 LIST＝∅且 maxf(t)＝0)，转 STEP1.

STEP4 从 LIST 中移走一个节点 i；寻找从节点 i 出发的所有可能的增广弧：

STEP4.1 对非饱和前向弧 (i, j)，若节点 j 没有标号(即 $\mathrm{pred}(j)=0$)，则对 j 进行标号，即令 $\mathrm{maxf}(j)=\min\{\mathrm{maxf}(i), u_{ij}-x_{ij}\}$，$\mathrm{pred}(j)=i$，并将 j 加入 LIST 中.

STEP4.2 对非空反向弧 (j, i)，若节点 j 没有标号(即 $\mathrm{pred}(j)=0$)，则对 j 进行标号，令 $\mathrm{maxf}(j)=\min\{\mathrm{maxf}(i), x_{ji}\}$，$\mathrm{pred}(j)=i$，并将 j 加入 LIST 中.

STEP5 转 STEP3.

算法的正确性与计算复杂性：从算法的过程可以看出，只要网络中存在增广路，算法就一定能找到增广路.因此，标号算法终止时，网络中一定不再含有增广路，根据增广路定理，该算法就求到了最大流.那么，该算法是否一定在有限步内终止呢？如果弧上的容量可以是无理数，可以举出例子说明标号算法不一定在有限步内终止；并且，这一增广路算法的极限过程得到的流的流值即使收敛，也不一定收敛到最大流(参见本章练习题20).如果所有弧上的容量都是正整数，根据整流定理，最大流 v 也是整数.实际上，由于割 $[\{s\}, V\backslash\{s\}]$ 中前向弧的条数最多为 n 条，因此最大流 v 的上界为 nU(U 表示网络中弧上的最大容量).由于每次增广至少使得流值增加1个单位，因此增广的次数不会超过 v 次，算法一定在有限步内终止.此外，由于每次增广最多需要对所有弧检查一遍，因此算法的复杂度为 $O(mv)$，或表示为 $O(mnU)$.可见，这一算法只是伪多项式时间算法，而不是多项式时间算法.

例 5.2 用 Ford-Fulkerson 算法计算如下网络中的最大流，每条弧上的两个数字分别表示容量和当前流量(图 5.2(a)).

第1次增广：

(1) 从 $s=1$ 开始标号，检查节点1相关的弧发现(1，3)和(1，4)为不饱和前向弧，因此节点3和4获得标号.此时 LIST=\{3，4\}，节点3的标号为(1，1)，节点4的标号为(1，1).

(2) 从 LIST 中取节点3，检查节点3相关的弧发现没有不饱和前向弧或非空反向弧.继续从 LIST 中取节点4，检查节点4相关的弧发现有不饱和前向弧(4，5)，因此节点 $t=5$ 获得标号.此时 LIST=$\{t\}$.节点5的标号为(4，min\{1，3\})=(4，1).

(3) 由于节点 $t=5$ 获得标号，已经找到一条增广路，因此进行增广.增广后网络上的流如图5.2(b)所示.

第2次增广：

(4) 从 $s=1$ 开始标号，检查节点1相关的弧发现(1，3)为不饱和前向弧，因此节点3获得标号.此时 LIST=\{3\}，节点3的标号为(1，1).

(5) 从 LIST 中取节点3，检查节点3相关的弧发现弧(3，4)为非空反向弧，因此节点

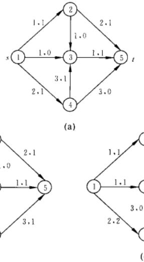


图 5.2 Ford-Fulkerson 标号算法的例子(例 5.2)

4 获得标号. 此时 LIST={4}，节点 4 的标号为(3，min{1，3})=(3，1).

(6) 从 LIST 中取节点 4，检查节点 4 相关的弧发现有不饱和前向弧(4，5)，因此节点 $t=5$ 获得标号. 此时 LIST={t}，节点 5 的标号为(4，min{1，3−1})=(4，1).

(7) 由于节点 $t=5$ 获得标号，已经找到一条增广路，因此进行增广. 增广后网络上的流如图 5.2(c)所示.

此时网络中不存在增广路，已经得到最大流，流值为 4. □

5.2.2 残量网络

下面介绍如何克服 Ford-Fulkerson 标号算法的缺点，更有效地找到增广路. 我们首先定义网络关于流的残量网络(residual network，又常称为增量网络或辅助网络). 我们在本章后面进行算法复杂度分析时，讨论中总是假定原网络满足这样的性质：弧 $(i,j)\in A$ 时，弧 $(j,i)\in A$. 这只要允许容量为 0 的弧仍然保留在网络中就可以了. 因此在复杂度上不会有本质的区别.

定义 5.7 对网络 $N=(s,t,V,A,U)$ 中给定的 s-t 可行流 $\boldsymbol{x}$，网络 N 关于流 $\boldsymbol{x}$ 的残量网络 $N(\boldsymbol{x})=(s,t,V,A(\boldsymbol{x}),U(\boldsymbol{x}))$，其中 $U(\boldsymbol{x})$ 定义如下：

$$u_{ij}(\boldsymbol{x})=u_{ij}-x_{ij}+x_{ji},\quad (i,j)\in A \tag{5.6}$$

其中 $u_{ij}(\boldsymbol{x})$ 称为弧(i,j)上的残留容量(residual capacity). 残量网络中的弧集 $A(\boldsymbol{x})$ 定义为 $\{(i,j)\mid u_{ij}(\boldsymbol{x})>0\}$.

这一定义实际上是给出了残量网络的构造方法. 根据定义，可以得到残量网络与原网

络中的流之间的关系：如果 v^* 是原网络中的最大流，则残量网络 $N(\boldsymbol{x})$ 中的最大流为 $v^* - v(\boldsymbol{x})$.

例 5.3　在如下的流网络 N 中(如图 5.3(a)所示)，每条弧上的两个数字分别表示容量和当前流量. 其残量网络 $N(\boldsymbol{x})$ 如图 5.3(b)所示. □

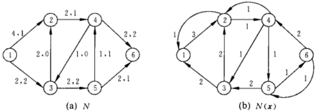


图 5.3　残量网络的例子(例 5.3)

于是，对于 $N(\boldsymbol{x})$ 中的任何一个 s-t 有向路 P，它一定对应于原网络 N 中的一条增广路，即可以通过沿 P 对当前流 $\boldsymbol{x}$ 进行增广，获得流值更大的 s-t 可行流 $\mathbf{y}$. 为了方便，我们也直接称残量网络中的 s-t 有向路为增广路. 沿 P 可以增广的最大流量等于 P 中所有弧上的残留容量中的最小值，我们称该最小值为 P 的残留容量.

5.2.3　最大容量增广路算法

Ford-Fulkerson 标号算法每次只是在所有增广路中随便地找一条进行增广，因此增广的次数可能很多. 一个自然的想法是：如果每次都找一条增广容量最大的增广路，则总的增广次数应当减少. 这样的算法称为最大容量增广路算法.

前面说过，如果 v^* 是原网络中的最大流，则残量网络 $N(\boldsymbol{x})$ 中的最大流为 $v^* - v(\boldsymbol{x})$. 根据流的分解定理，在残量网络中可以找到不超过 m 条从源点到汇点的有向路(增广路)，使得其残留容量之和为 $v^* - v(\boldsymbol{x})$. 因此，残量网络 $N(\boldsymbol{x})$ 中的最大容量增广路的残留容量不会小于 $(v^* - v(\boldsymbol{x}))/m$.

现在考虑从可行流 $\boldsymbol{x}$ 开始的连续 $2m$ 次最大容量增广. 如果每次增广的流量都不小于 $(v^* - v(\boldsymbol{x}))/(2m)$，则 $2m$ 次增广后一定已经达到了最大流；如果某次增广的流量小于 $(v^* - v(\boldsymbol{x}))/(2m)$，则说明此时残量网络中的最大流小于 $(v^* - v(\boldsymbol{x}))/2$. 因此，每经过连续 $2m$ 次最大容量增广，残量网络中的最大容量增广路的残留容量至少下降一半. 由于任何增广路的残留容量不会超过 U(U 表示网络中弧上的最大容量)且至少为 1(这里假设只考虑整数容量网络)，因此最多经过 $O(2m\log U) = O(m\log U)$ 次最大容量增广，一定已经达到了最大流.

可见，最大容量增广路算法将总的增广次数从 Ford-Fulkerson 标号算法的 $O(nU)$ 降为了 $O(m\log U)$ 次，因此是一种多项式时间算法. 但由于每次需要在残量网络中寻找最大容量增广路，因此每次增广的时间花费增加了. 下面我们介绍一个算法，其增广次数也是

$O(m\log U)$次，而且每次在残量网络中寻找增广路的时间花费也很低.

5.2.4 容量变尺度算法

为了使总的增广次数减少，容量变尺度算法每次都找一条增广容量"充分大"的增广路，但不一定最大. 为此，引入如下概念.

对网络 $N=(s,t,V,A,U)$中给定的 s-t 可行流 $\boldsymbol{x}$，网络 N 关于流 $\boldsymbol{x}$ 的 Δ-残量网络 $N(\boldsymbol{x},\Delta)$ 是其残量网络 $N(\boldsymbol{x})$的子网络，它由残留容量不小于 Δ 的弧组成.

可见，对于整数 s-t 网络，$N(\boldsymbol{x},1)=N(\boldsymbol{x})$. 在 $N(\boldsymbol{x},\Delta)$中，每次增广的流量至少为 Δ. 我们从 $\Delta=2^{\lfloor\log U\rfloor}$开始，每经过一定次数的增广后将 Δ 的值减少一半，直到 $\Delta=1$. 即算法为：

容量变尺度算法

STEP0　置初始可行流 $\boldsymbol{x}$(如零流)；$\Delta=2^{\lfloor\log U\rfloor}$.

STEP1　若 $\Delta<1$，结束，已经得到最优解；否则进行下一步.

STEP2　构造 Δ-残量网络 $N(\boldsymbol{x},\Delta)$.

STEP3　若 $N(\boldsymbol{x},\Delta)$不存在 s-t 路，继续下一步；否则从 $N(\boldsymbol{x},\Delta)$找到一条 s-t 路 P，沿 P 对当前流进行增广，并重复 STEP3.

STEP4　$\Delta=\Delta/2$；转 STEP1.

复杂度分析：Δ 的值总共有 $O(\log U)$种取法. 我们下面说明在给定的 Δ 下，最多执行 $2m$ 次增广. 由于在 2Δ-残量网络中的最大流不会超过 $2m\Delta$. 因为 Δ-残量网络中的增广路的容量至少为 Δ，因此增广次数不会超过 $2m$.

因此，算法总的增广次数是 $O(m\log U)$次，而每次增广路的查找可以采用前面介绍的标号算法，即复杂度为 $O(m)$. 因此，算法总复杂度为 $O(m^2\log U)$.

5.3 最短增广路算法

与最大容量增广路算法对称，一个自然的想法是：如果每次都找一条包含弧数最少的增广路(称为最短增广路)，则算法效果如何？这样的算法称为最短增广路算法(可见，这种算法也是一种增广路算法). 可以证明，最短增广路算法最多经过 $O(mn)$次增广后终止. 对于这个特殊的最短路问题，我们可以很容易构造最短路算法在 $O(m)$的时间内找到一条最短增广路(例如可以采用从节点 s 或 t 开始的广度优先搜索方法，请读者思考)，因此这种实现方法的复杂度为 $O(m^2n)$. 利用特殊的标号过程，实际上可以在 $O(n)$的平均时间内找到一条最短增广路. 进一步，利用更为复杂的数据结构，可以在 $O(\log n)$的平均时间内找到一条最短增广路. 因此，这些实现方法的复杂度分别为 $O(mn^2)$和 $O(mn\log n)$. 从目前已有

的理论分析和计算经验来看，最短增广路算法是所有增广路算法中效果最好的算法，因此本节对复杂度为$O(mn^2)$的算法进行详细介绍．下面我们首先介绍在本节以及后续几节中极为重要的一个概念：距离标号．

5.3.1 距离标号

定义 5.8 对于一个残量网络$N(\boldsymbol{x})$，如果一个函数d将节点集合V映射到非负整数集合，则称d是关于残量网络$N(\boldsymbol{x})$的距离函数(distance function)，$d(i)$称为节点i的距离标号(distance label)．如果距离函数d满足：

(1) $d(t)=0$，

(2) 对$N(\boldsymbol{x})$中的任意一条弧(i,j)有$d(i)\leqslant d(j)+1$，

则称距离函数d关于流$\boldsymbol{x}$是有效的(valid)，或称距离标号是有效的．如果任意一个节点的距离标号正好等于残量网络中从该节点到汇点(节点t)的所有有向路中弧数最少的有向路所包含的弧数(当令所有弧的长度为1时，这就是从该节点到汇点的最短路路长，因此后面直接称为最短路路长)，则称距离函数d关于流$\boldsymbol{x}$是精确的(exact)，或称距离标号是精确的．

此外，如果对$N(\boldsymbol{x})$中的某一条弧(i,j)有$d(i)=d(j)+1$，则称弧(i,j)为允许弧(admissible arc)．一条s-t有向路如果完全由允许弧组成，则该有向路称为允许路(admissible path)．

例 5.4 假设残量网络如图5.4所示，节点旁边的数值表示距离标号，节点1为源s，5为汇t．则图5.4(a)中的标号是有效的，图5.4(b)中的标号是精确的． □

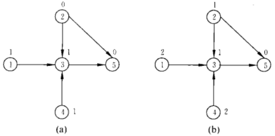


图5.4 距离标号的例子(例5.4)

根据以上定义，可以得到以下结论．

引理 5.3 若距离函数d是有效的，则：

(1) $d(i)$是残量网络$N(\boldsymbol{x})$中从节点i到节点t的最短有向路路长的下界．

(2) 如果$d(s)\geqslant n$，则残量网络$N(\boldsymbol{x})$中从节点s到节点t没有有向路(增广路)．

(3) 允许路是残量网络$N(\boldsymbol{x})$中的最短增广路． □

对于一个残量网络 $N(\boldsymbol{x})$，如何确定其精确的距离标号呢？我们可以从汇点(节点 t)开始，对 $N(\boldsymbol{x})$沿反向弧进行广度优先搜索，确定每个节点的精确的距离标号. 这一过程的复杂度为 $O(m)$. 可以看出，一个节点的精确的距离标号实际上表示的是从该节点到汇点(节点 t)的最短路路长，也就是说对所有节点按照最短路路长进行了层次划分.

5.3.2 最短增广路算法

最短增广路算法每次都找一条最短增广路，即残量网络 $N(\boldsymbol{x})$中的允许路进行增广. 基于距离标号，具体算法可以描述如下：

最短增广路算法

STEP0(预处理) 置初始可行流 $\boldsymbol{x}$ 为零流；计算精确的距离函数 d；令当前节点 $i=s$.

STEP1 若 $d(s)<n$，继续下一步；否则结束，已经得到最优解 $\boldsymbol{x}$.

STEP2 如果存在节点 i 的某条出弧(i, j)为允许弧，则转 STEP3；否则转 STEP4.

STEP3 令 $\text{pred}(j)=i$，再令 $i=j$；若 $i=t$，则找到了一条增广路，进行增广，修改残量网络，并令当前节点 $i=s$. 转 STEP1.

STEP4 修改标号(重新标号)：当$\{(i, j)\in A(\boldsymbol{x})\mid u_{ij}(\boldsymbol{x})>0\}\neq\varnothing$时，令 $d(i)=\min\{d(j)+1\mid(i, j)\in A(\boldsymbol{x})$且 $u_{ij}(\boldsymbol{x})>0\}$，否则令 $d(i)=n$；且当 $i\neq s$ 时再令 $i=\text{pred}(i)$. 转 STEP1.

STEP3 表示从当前节点沿一条允许弧“前进”一步到下一个节点，因此可以称为“前进步”；STEP4 前半部分表示当没有从当前节点 i 出发的允许弧时要修改标号，如果此时残量网络中没有从节点 i 出发的弧，则表明节点 i 再也不可能出现在任何 s-t 增广路中，所以可以令 $d(i)=n$. STEP4 后半部分表示当当前节点 $i\neq s$ 时“回退”一步，因此可以称 STEP4 中当 $i\neq s$ 时所进行的操作为“回退步”. 为了说明算法确实是正确的，我们首先证明一些基本的结论.

引理 5.4 在最短增广路算法的执行过程中，距离标号始终保持是有效的.

证明 采用归纳法证明. 在预处理阶段，构造的距离标号是有效的. 我们只需要证明算法的一次增广操作和一次重新标号操作不会改变距离标号的有效性.

一次沿允许路的增广操作可能引起的残量网络的弧的变化只有两种情况：允许路上的弧(i, j)可能退出残量网络，这不会影响(i, j)弧的端点上的距离标号的有效性；此外，(j, i)弧可能进入残量网络，由于(i, j)弧是允许弧，因此 $d(i)=d(j)+1$，所以 $d(j)=d(i)-1<d(i)$，这也不会影响(j, i)弧的端点上的距离标号的有效性. 因此算法的一次增广操作不会改变距离标号的有效性.

设一次对节点 i 的重新标号操作将 $d(i)$修改为 $d'(i)=\min\{d(j)+1\mid(i, j)\in A(\boldsymbol{x})$且

$u_{ij}(\boldsymbol{x})>0\}$. 显然，这不会破坏任意的$(i, j)$弧的端点上的距离标号的有效性. 由于重新标号时节点 i 在残量网络中没有允许弧，也就是说对于任意的(i, j)弧满足 $d(i)<d(j)+1$. 因此 $d'(i)>d(i)$，即标号严格增加. 对于任意的(j, i)弧，$d(j)\leqslant d(i)+1<d'(i)+1$，因此也不会破坏任意的$(j, i)$弧的端点上的距离标号的有效性. 故算法的一次重新标号操作不会改变距离标号的有效性. □

由于距离函数是有效的，算法结束时残量网络 $N(\boldsymbol{x})$ 中从节点 s 到节点 t 没有允许路，即残量网络中不存在增广路，所以算法确实求到了最大流.

例 5.5 用最短增广路算法计算网络图 5.5(a)中的最大流(弧上数字表示容量).

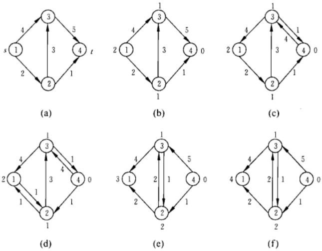


图 5.5 最短增广路算法的例子(例 5.5)

为了直观，我们直接把节点的距离标号 $d(i)$标在节点 i 旁边. 首先进行预处理，得到残量网络图 5.5(b)，当前节点为 $s=1$.

从 1 出发，找到允许路(1, 3)，(3, 4). 增广 4 个单位流量，得到残量网络图 5.5(c).

重新从 1 出发，找到允许路(1, 2)，(2, 4). 增广 1 个单位流量，得到残量网络图 5.5(d).

重新从 1 出发，找到部分允许路(1, 2)，但节点 2 出发没有允许弧. 因此将节点 2 的标号修改为 2，并退回一步，即回到节点 1. 但节点 1 出发没有允许弧，因此将节点 1 的标号修改为 3.

继续从 1 出发，找到允许路(1, 2)，(2, 3)，(3, 4). 增广 1 个单位流量，得到残量网络图 5.5(e).

重新从 1 出发，找不到允许路，因此按照算法应当将节点 1 的标号进行修改. 但残量网络中从 1 出发的弧为空集合，我们不妨令节点 1 的标号为 4，得到残量网络图 5.5(f).

此时算法结束，得到最大流流值为 6. □

5.3.3 复杂度分析

为了分析算法的计算复杂性，我们首先必须确定算法的一些具体实现细节. 在算法的每次迭代中，我们要从一个节点出发找到从该节点出发的一条允许弧(或证明从该节点出发不存在允许弧). 一般可以采用当前弧(current-arc)数据结构，即对每个节点 i，算法用链表 $A(i)$记录该节点出发的所有弧，这些弧的顺序可以任意，但一旦确定以后在算法执行过程中就不再改变. 对每个节点 i，用一个指针指向下一条将要被检索的弧，这条弧称为当前弧. 算法开始时，当前弧为 $A(i)$中的第一条弧. 在查找从该节点出发的允许弧时，总是从当前弧开始判别：如果当前弧是允许弧，则返回这条弧；否则令 $A(i)$中的下一条弧为当前弧. 如此下去，直到找到一条允许弧或判别完 $A(i)$的最后一条弧为止. 可以看出，如果 $A(i)$中的一条弧在以前的迭代中不是允许弧，则在节点 i 的标号改变之前它仍然不是允许弧(请读者作为练习完成其证明). 因此，如果判别完 $A(i)$的最后一条弧仍然没有找到允许弧，则说明 $A(i)$中不存在允许弧；这时，算法必须执行对该节点的重新标号操作，并重置该节点的当前弧为 $A(i)$中的第一条弧. 由于对节点 i 的一次检查和一次重新标号操作分别需要对 $A(i)$检索一遍，因此复杂度为 $O(|A(i)|)$. 这样我们得到如下结论.

引理 5.5 在最短增广路算法的执行过程中，如果每个节点的重新标号操作不超过 k 次，则算法查找允许弧和重新标号操作的总复杂度为 $O(\sum_{i\in V} k\ |A(i)|) = O(km)$. □

例 5.6 假设从节点 1 出发的弧为 $A(1)=\{(1,2),(1,3),(1,4),(1,5),(1,6)\}$，如图 5.6 所示. 弧上的数值表示残留容量，节点旁边的数值表示距离标号. 初始化时当前弧为(1, 2). 由于它不是允许弧(因为残留容量为零，所以可以认为它根本不属于当前的残量网络)，令当前弧为(1, 3). 由于它仍然不是允许弧，令当前弧为(1, 4)，它是一条允许弧，并继续保持为当前弧. □

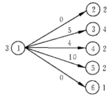


图 5.6 当前弧结构(例 5.6)

下面我们再证明其他一些有用的引理.

引理 5.6 如果算法对任何节点最多进行 k 次重新标号，则算法最多 $km/2$ 次使网络中的弧饱和(即弧上的残留容量减少为零). 因此，最短增广路算法最多执行 $km/2$ 次增广.

证明 首先证明在对弧(i, j)进行的两次连续的并使之饱和的增广之间，其两个端点的距离标号都至少增加了 2 个单位. 在前一次增广时，由于弧(i, j)是允许弧，因此距离标号 d 满足 $d(i)=d(j)+1$. 在后一次增广之前，必须有一定的流量沿弧(j, i)增广过. 沿弧(j, i)增广时弧(j, i)是允许弧，因此距离标号 d'满足 $d'(j)=d'(i)+1$. 在后一次增广之时，由于弧(i, j)是允许弧，因此距离标号 d''满足 $d''(i)=d''(j)+1$. 由于距离标号是非降的，于是有

$$d''(i)=d''(j)+1\geqslant d'(j)+1=d'(i)+2\geqslant d(i)+2,$$
$$d''(j)\geqslant d'(j)=d'(i)+1\geqslant d(i)+1=d(j)+2.$$

如果算法对任何节点最多进行 k 次重新标号，则算法对任何弧最多执行 $k/2$ 次使之饱和的增广. 所以对于所有弧而言，算法最多执行 $km/2$ 次使之饱和的增广. 由于每次增广至少使一条弧饱和，因此最短增广路算法最多执行 $km/2$ 次增广. □

引理 5.7 最短增广路算法在执行过程中，最多使每个节点的标号增加 n 次. 因此，最短增广路算法最多执行 $nm/2$ 次增广.

证明 最短增广路算法在执行过程中，每次对一个节点的重新标号操作使得该节点的标号增加至少一个单位. 因此对一个节点的标号增加 n 次后，其标号将至少为 n，此时它不会再出现在任何增广路中，因此算法不再选择它探索允许路. 所以，最短增广路算法在执行过程中，最多使每个节点的标号增加 n 次. 因此，对所有节点的重新标号操作不超过 n^2.

因此，最短增广路算法最多执行 $nm/2$ 次增广. □

此外，该引理说明，算法对所有节点的重新标号操作不超过 n^2 次.

定理 5.5 最短增广路算法的复杂度为 $O(n^2m)$.

证明 每个节点的重新标号操作不超过 n 次，所以算法查找允许弧和重新标号操作的总复杂度为 $O(nm)$. 由于每次增广的复杂度为 $O(n)$，因此增广花费的总时间为 $O(n^2m)$. 由于每次回退操作对一个节点重新标号，因此算法执行回退的次数为 $O(n^2)$. 综上所述，最短增广路算法的复杂度为 $O(n^2m)$. □

5.4 一般的预流推进算法

5.4.1 一般的预流推进算法

增广路算法的固有缺陷在于它找到增广路以后需要马上沿增广路对流进行增广，而每一次增广可能需要对最多 $n-1$ 条弧操作，因此每一次增广复杂度为 $O(n)$. 在有些情况下，这是极其浪费时间的. 看下面一个极端的例子.

例 5.7 如图 5.7 所示的网络中，$s=1,t=20$. 弧 $(1,2),(2,3),\cdots,(8,9)$ 上的容量充分大(如等于 10)，所有其他弧(节点 9 的出弧和节点 20 的入弧)上的容量均为 1. 无论采用何种增广路算法，都会找到 10 条增广路，每条路长为 10，容量为 1. 因此，总共需要 10 次增广，每次增广 1 个单位，需要对 10 条弧进行操作. 然而，10 条增广路中的前 9 个节点(前 8 条弧)是完全一样的，能否直接将前 8 条弧的流量增广 10 个单位，而只对后面长为 2 的不同的有向路单独操作呢？这样可以节省许多计算时间. 这就是预流推进算法(preflow push algorithm)的思想. 也就是说，预流推进算法关注于对每一条弧的操作和处理，而不必一次一定处理一条增广路. □

一般把沿一条弧进行流量增广的基本操作称为一次推进(push). 在推进过程中，流一

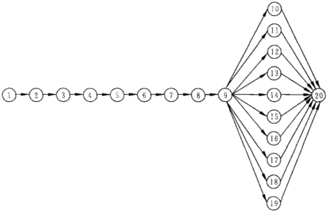


图 5.7 增广路算法的一个极端例子(例 5.7)

定满足容量约束(5.2a)，但一般不满足流量平衡方程(5.4)，因此只是一个伪流. 此外，从每个节点(除 s，t 外)流出的流量之和总是小于等于流入该节点的流量之和，我们称这样的伪流为预流(preflow). 这就是这类算法称为预流推进算法的原因. 下面首先给出预流的严格定义.

定义 5.9 流网络 $N=(s,t,V,A,U)$上的一个预流 $\boldsymbol{x}$ 是指从 N 的弧集 A 到实数集合$\mathbb{R}$的一个函数，使得对每个节点 i 都满足如下条件：

$$e(i)\geqslant 0,\quad i\neq s,\ t, \tag{5.7}$$

$$0\leqslant x_{ij}\leqslant u_{ij},\quad \forall\,(i,\ j)\in A, \tag{5.8}$$

其中 $e(i)=\sum\limits_{j:(j,\ i)\in A}x_{ji}-\sum\limits_{j:(i,\ j)\in A}x_{ij}\ (\forall\, i\in V)$，称为预流 $\boldsymbol{x}$ 在节点i 上的赢余(excess). $e(i)>0$ 的节点 $i(i\neq s,\ t)$ 称为活跃的(active).

对流网络 $N=(s,t,V,A,U)$上的一个预流 $\boldsymbol{x}$，如果存在活跃节点，则说明该预流是不可行的. 预流推进算法就是要选择活跃节点，并通过把一定的流量推进到它的邻居(回忆第 1 章中对邻居的定义，即如果弧$(i,\ j)\in A$，我们称 j 是 i 的一个邻居)，尽可能地将正的赢余减少为 0. 如果当前活跃点有多个邻居，那么首先推进到哪个邻居呢？由于算法最后的目的是尽可能将流推进到汇(节点 t)，因此算法总是首先寻求把流量推进到它的邻居中距离节点 t 最近的节点. 由于每个节点的距离标号可以表示其与节点 t 的距离，因此算法总是将流量沿着允许弧推进. 如果当前活跃点出发没有允许弧，则增加该节点的距离标号，使得从当前活跃点出发至少含有一条允许弧.

下面给出一般的预流推进算法的基本框架：

一般的预流推进算法

STEP0(预处理)　置初始可行流 $\boldsymbol{x}$ 为零流；对节点 s 的每条出弧 (s, j) 令 $x_{sj}=u_{sj}$；对任意的 $i\in V$ 计算精确的距离标号 $d(i)$；令 $d(s)=n$.

STEP1　如果残量网络中不存在活跃节点，结束，已经得到最优解；否则继续.

STEP2　在网络中选取活跃节点 i；如果存在节点 i 的某条出弧 (i, j) 为允许弧，则将 $\min\{e(i), u_{ij}(\boldsymbol{x})\}$ 个单位的流从节点 i 推进到节点 j；否则令 $d(i)=\min\{d(j)+1 \mid (i, j)\in A(\boldsymbol{x})$ 且 $u_{ij}(\boldsymbol{x})>0\}$. 转 STEP1.

可见，算法的每次迭代是一次推进操作(STEP2 的前面部分)或者一次重新标号操作(STEP2 的后面部分). 对于推进操作，如果推进的流量等于弧上的残留容量，则称为饱和推进(saturating push)，否则称为非饱和推进(nonsaturating push). 当算法终止时，网络中不含有活跃节点，因此只有节点 s，t 的赢余为非零数. 所以，此时得到的预流实际上已经是一个可行流. 又由于在算法预处理阶段已经令距离标号 $d(s)=n$，而距离标号在计算过程中不会减少，因此算法在计算过程中可以保证网络中永远不会有增广路存在. 根据增广路定理，算法终止时一定得到了最大流.

例 5.8　用预流推进算法计算如图 5.8 网络(a)中的最大流(弧上数字表示容量).

节点 $s=1$，$t=4$，因此只有节点 2 和节点 3 可以是活跃节点. 为了直观，我们直接把节点的赢余 $e(i)$(第 1 个数字)和距离标号 $d(i)$(第 2 个数字)标在节点 i 旁边.

首先进行预处理，得到残量网络(b).

此时节点 2，3 都是活跃节点. 假设选择节点 2，沿弧(2，4)推进 1 个单位流量，得到残量网络(c).

此时节点 2，3 仍然都是活跃节点. 假设继续选择节点 2，由于没有从节点 2 出发的允许弧，所以将节点 2 的标号修改为 2.

此时节点 2，3 仍然都是活跃节点. 假设选择节点 3，沿弧(3，4)推进 4 个单位流量，得到残量网络(d).

此时只有节点 2 是活跃节点. 沿弧(2，3)推进 2 个单位流量，得到残量网络(e).

此时只有节点 3 是活跃节点. 沿弧(3，4)推进 1 个单位流量，得到残量网络(f).

此时只有节点 3 是活跃节点. 由于没有从节点 3 出发的允许弧，所以将节点 3 的标号修改为 3.

此时只有节点 3 是活跃节点. 沿弧(3，2)推进 1 个单位流量，得到残量网络(g).

此时只有节点 2 是活跃节点. 由于没有从节点 2 出发的允许弧，所以将节点 2 的标号修改为 4.

此时只有节点 2 是活跃节点. 沿弧(2，3)推进 1 个单位流量，得到残量网络(h).

此时只有节点 3 是活跃节点. 由于没有从节点 3 出发的允许弧，所以将节点 3 的标号

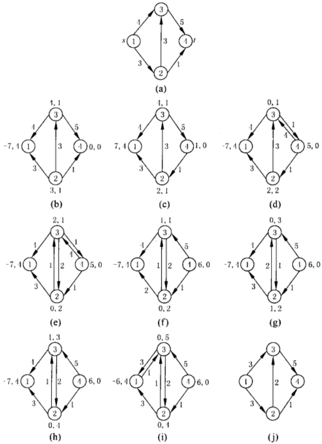


图 5.8 预流推进算法的例子(例 5.8)

修改为 5.

此时只有节点 3 是活跃节点.沿弧(3, 1)推进 1 个单位流量，得到残量网络(i).

此时没有活跃节点，算法结束，得到最大流如图(j).最大流流值为 6. □

5.4.2 复杂度分析

我们首先证明一些基本的结论.

引理 5.8 在预流推进算法的执行过程中，距离标号始终保持是有效的.

证明 采用归纳法证明.在预处理阶段，构造的距离标号是有效的.我们只需要证明算法的一次推进操作和一次重新标号操作不会改变距离标号的有效性.

一次沿允许弧(i, j)弧的推进操作可能引起的残量网络的弧的变化只有两种情况:$(i,$

j)弧可能退出残量网络(饱和推进)，这不会影响(i, j)弧的端点上的距离标号的有效性；(j, i)弧可能进入残量网络，由于(i, j)弧是允许弧，因此$d(i)=d(j)+1$，所以$d(j)=d(i)-1<d(i)$,这也不会影响(i, j)弧或(j, i)弧的端点上的距离标号的有效性.因此算法的一次推进操作不会改变距离标号的有效性.

设一次对节点i的重新标号操作将$d(i)$修改为$d'(i)=\min\{d(j)+1 \mid (i, j)\in A(\boldsymbol{x})$且$u_{ij}(\boldsymbol{x})>0\}$.显然，这不会破坏任意的$(i, j)$弧的端点上的距离标号的有效性.由于重新标号时节点$i$在残量网络中没有允许弧，也就是说对于任意的$(i, j)$弧满足$d(i)<d(j)+1$.因此$d'(i)>d(i)$，即标号严格增加.对于任意的$(j, i)$弧，$d(j)\leqslant d(i)+1<d'(i)+1$，因此也不会破坏任何$(j, i)$弧的端点上的距离标号的有效性.故算法的一次重新标号操作不会改变距离标号的有效性. □

引理5.9　在预流推进算法的执行过程中，在残量网络上，从任何一个活跃节点到源点s都存在一条有向路.

证明　对于任何一个预流$\boldsymbol{x}$有$e(s)\leqslant 0$，$e(i)\geqslant 0(i\neq s)$.根据流的分解定理，$\boldsymbol{x}$在原网络N中一定可以分解为一系列路流和圈流之和，并且这些路流总是从s到t或一个活跃节点.由于任何圈流以及从s到t的路流不会使得任何节点$i(i\neq s, t)$产生赢余，因此对于活跃节点i，$\boldsymbol{x}$在原网络N中分解的路流一定有一个是从s到活跃节点i.于是，在残量网络中，一定有一条从i到节点s的有向路. □

引理5.10　在预流推进算法的执行过程中，$d(i)<2n$.因此，修改距离标号的总次数不超过$2n^2$.

证明　当算法最后一次对节点i进行重新标号时，前面的引理表明一定有一条从i到节点s的有向路.由于该有向路最多包含$n-1$条弧，并且$d(s)=n$，因此$d(i)\leqslant d(s)+n-1<2n$.

于是，算法对每个节点i修改距离标号的次数不超过$2n$，对所有节点修改距离标号的总次数不超过$2n^2$. □

为了分析算法的计算复杂性，我们首先必须确定算法的一些具体实现细节.在算法的每次迭代中，我们要找到一个活跃节点，然后找到从该节点出发的一条允许弧(或证明从该节点出发不存在允许弧).找到一个活跃节点比较容易，我们只要在算法中使用一个链表LIST记录所有活跃节点，每次从中取出一个即可，因此每次迭代找到一个活跃节点的复杂度为$O(1)$.

为了找到从一个活跃节点出发的一条允许弧(或证明从该节点出发不存在允许弧)，与5.3.3节一样，可以采用当前弧(current-arc)数据结构.这样我们仍然得到如下结论：

引理5.11　在预流推进算法的执行过程中，如果每个节点的重新标号操作不超过k次，则算法查找允许弧和重新标号操作的总复杂度为$O(\sum_{i\in V}k\mid A(i)\mid)=O(km)$. □

下面我们再证明其他一些有用的引理.

引理 5.12 如果算法对任何节点最多进行 k 次重新标号，则算法最多执行 $km/2$ 次饱和推进. 因此，预流推进算法最多执行 nm 次饱和推进.

证明 首先证明在对弧(i,j)进行的两次连续的饱和推进之间，其两个端点的距离标号都至少增加了 2 个单位. 在前一次饱和推进时，由于弧(i,j)是允许弧，因此距离标号 d 满足 $d(i)=d(j)+1$. 在后一次饱和推进之前，必须有一定的流量沿弧(j,i)推进过. 沿弧(j,i)推进时弧(j,i)是允许弧，因此距离标号 d'满足 $d'(j)=d'(i)+1$. 在后一次饱和推进之时，由于弧(i,j)是允许弧，因此距离标号 d''满足 $d''(i)=d''(j)+1$. 由于距离标号是非降的，于是有

$$d''(i)=d''(j)+1\geqslant d'(j)+1=d'(i)+2\geqslant d(i)+2,$$
$$d''(j)\geqslant d'(j)=d'(i)+1\geqslant d(i)+1=d(j)+2.$$

如果算法对任何节点最多进行 k 次重新标号，则算法对任何弧最多执行 $k/2$ 次饱和推进. 所以对于所有弧而言，算法最多执行 $km/2$ 次饱和推进. 因此，预流推进算法最多执行 nm 次饱和推进. □

引理 5.13 预流推进算法最多执行 $O(n^2m)$次非饱和推进.

证明 令 I 是所有活跃节点的集合，$\Phi=\sum_{i\in I}d(i)$. 显然，算法开始时(预处理以后)，$\Phi\leqslant 2n^2$ (因为所有的距离标号均不超过 $2n$)；算法结束时，$\Phi=0$. 在算法对活跃节点 i 操作的过程中，有 3 种可能：

(1) 算法没有找到允许弧，进行一次重新标号操作. 也就是说，节点 i 的标号增加 $\varepsilon\geqslant 1$ 个单位，由此引起的 Φ 的增加至多为 ε 个单位. 因为所有的距离标号均不超过 $2n$，因此整个算法由于重新标号操作引起的 Φ 的增加至多为 $2n^2$ 个单位.

(2) 算法找到允许弧(i,j)，进行一次饱和推进操作. 此时，节点 j 可以从非活跃节点变成为活跃节点，且节点 i 可能仍保持为活跃节点. 因此，由此引起的 Φ 的增加至多为 $d(j)\leqslant 2n$ 个单位. 因为所有的饱和推进不超过 mn 次，因此整个算法由于饱和推进操作引起的 Φ 的增加至多为 $2mn^2$ 个单位.

(3) 算法找到允许弧(i,j)，进行一次非饱和推进操作. 此时，节点 j 可以从非活跃节点变成为活跃节点，但节点 i 一定转变为非活跃节点. 因此，由此引起的 Φ 的减少至少为 $d(i)-d(j)=1$ 个单位.

以上分析表明，整个算法过程中 Φ 的增加值最大不超过 $2n^2+2n^2+2mn^2=O(n^2m)$个单位. 由于 Φ 永远是非负的，且算法结束时 $\Phi=0$，因此预流推进算法最多执行 $O(n^2m)$次非饱和推进. □

定理 5.6 一般的预流推进算法的计算复杂度为 $O(n^2m)$.

证明 根据前面的引理，所有查找允许弧和重新标号操作可以在 $O(nm)$时间内完成. 此外，算法最多进行 $O(n^2m)$次推进操作，由于每次推进操作可以在 $O(1)$时间内完成，因此一般的预流推进算法的计算复杂度为 $O(n^2m)$. □

5.5 最高标号预流推进算法

5.5.1 最高标号预流推进算法

第1个预流推进算法是由 Karzanov 于 1974 年提出的，复杂度为 $O(n^3)$，这在当时的所有算法中是复杂度最低的多项式时间算法. 但当时的算法采用的工具不是距离标号，而是分层网络的概念. 距离标号可以看成是分层网络概念的推广，基于这一概念，20 世纪 80 年代末、90 年代初人们提出了一般的预流推进算法和大量的具体实现方案. 下面我们具体地介绍其中一种在大量实际计算中被认为是最有效的实现方案，它被称为最高标号预流推进算法(highest-label preflow-push algorithm)，是由 Goldberg 和 Tarjan 于 1986 年提出来的，Cheryian 和 Maheshwari 于 1989 年证明其复杂度为 $O(n^2m^{1/2})$.

在一般的预流推进算法中，每当选定了一个活跃节点后，如果执行的是一次饱和推进，则该节点仍然是活跃节点，但紧接着的下一次迭代可能选定另一个新的活跃节点. 我们可以在算法中增加一条规则，要求紧接着的下一次迭代尽可能选择紧接上次的活跃节点，直到该节点的赢余减少到 0 或必须对它进行重新标号以后为止. 我们以后总是讨论采用这样的规则的预流推进算法，并把针对同一节点的这些连续推进称为对该节点的一次检查.

从上面对一般的预流推进算法的复杂度分析中可以看出，算法的瓶颈操作在于非饱和推进. 最高标号预流推进算法的思想是从具有最大距离标号的活跃节点开始预流推进. 之所以按照这样的顺序做，一个直观的想法是：使得距离标号较小的活跃节点累积尽可能多的来自距离标号较大的活跃节点的流量，然后对累积的盈余进行推进，这样可能会减少非饱和推进的次数.

那么，如何以较小的代价实现每次从最高标号开始预流推进呢？我们可以在算法中用一个链表 LIST(k)记录距离标号为 k 的所有活跃节点($k=1, 2, \cdots, 2n-1$)，并用一个变量 $level$ 记录当前的活跃节点中最大可能的距离标号(上界). 这样，为了找到最高标号的活跃节点，只要依次扫描 LIST($level$)，LIST($level-1$)，…，直到找到一个非空的 LIST(p)为止. 此时令 $level=p$，并从 LIST(p)中选取任意一个节点即可. 每当某个活跃节点的距离标号增加时，我们只需要将新标号的值赋给 $level$，并将该活跃节点放到 LIST($level$)中. 从 5.4 节知道，距离标号的增加量最多不超过 $2n^2$，因此 $level$ 的增加量最多不超过 $2n^2$，所以 $level$ 的减少量最多也就不超过 $n+2n^2$. 于是，对链表进行操作以便找到最高标号节点的总复杂度为 $O(n^2)$. 这一操作不会成为算法的新的瓶颈.

5.5.2 算法的复杂度分析

如果算法连续经过对 n 个节点的检查而中间没有对任何节点进行距离标号的修改操

作，则赢余一定已经达到了 s 或 t 点，因此求到了最大流，算法终止. 由于标号的修改次数不超过 $2n^2$ 次，每两次相邻的标号修改操作之间对节点的检查不超过 n 次，而每次节点检查执行非饱和推进的次数不超过 1 次，这样我们就证明了最高标号预流推进算法执行非饱和推进的总次数不超过 $O(n^3)$ 次，因此算法的复杂度为 $O(n^3)$. 实际上，下面的讨论说明，我们可以证明更好的结果.

在 5.3.3 节介绍的当前弧数据结构中，对每一个节点至多只有一条弧同时为当前弧和允许弧，我们记所有这些弧的集合为 F. 显然，F 最多含有 $n-1$ 条弧，每个节点最多有一条出弧，并且不可能含有任何圈. 因此，F 是一个森林，我们称它为当前森林(current forest). 在 F 中，记所有后代(直接或间接后继，包括该节点本身)的集合为 $D(i)$. 根据距离标号的性质，$D(i)$ 中任何节点的标号不会小于节点 i 的标号 $d(i)$. 如果节点 i 是活跃节点，且除节点 i 以外 $D(i)$ 中不含有活跃节点，则称节点 i 是一个最大活跃节点. 显然，两个最大活跃节点不可能含有同一个后代节点. 进一步分析可以看出，最高标号预流推进算法实际上总是从一个最大活跃节点开始进行预流推进.

例 5.9　假设当前森林 F 如图 5.9 所示，节点旁边的数值表示赢余量，则：$D(1)=\{1\}$，$D(2)=\{2\}$，$D(3)=\{1,3,4\}$，$D(4)=\{4\}$，$D(5)=\{1,2,3,4,5\}$，等等. 其中，只有节点 2，4，8 为最大活跃节点. □

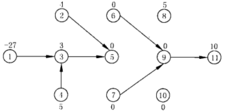


图 5.9　当前森林(数据结构)

记当前森林中最大活跃节点的集合为 H，并令 $\Phi=\sum_{i\in H}\Phi(i)$，其中 $\Phi(i)=\max\{0,K+1-|D(i)|\}$，$K$ 为一个待定参数. 显然，$0\leqslant\Phi(i)\leqslant K$. 下面讨论算法的各种操作对 Φ 值的影响，有 4 种可能：

(1) 对最大活跃节点 i，沿当前(允许)弧 (i,j) 执行一次非饱和推进. 推进后，当前弧 (i,j) 仍然为允许弧，所以当前森林不会发生改变. 但节点 i 变为非活跃节点，而节点 j 成为活跃节点，因此也可能成为新的最大活跃节点. 由于 $|D(i)|<|D(j)|$，所以当 $|D(i)|\leqslant K$ 时，$\Phi(i)+\Phi(j)$ 至少减少 1 个单位；否则，$\Phi(i)+\Phi(j)$ 保持不变.

(2) 对最大活跃节点 i，沿当前(允许)弧 (i,j) 执行一次饱和推进. 推进后，当前弧 (i,j) 不再是允许弧(因为残留容量为 0)，所以从当前森林中退出. 但节点 i 仍为最大活跃节点，同时节点 j 成为活跃节点，因此也可能成为新的最大活跃节点. 所以此时 Φ 最多可能增加 K 个单位.

(3) 对最大活跃节点 i，重新进行距离标号. 此时，说明从 i 出发没有允许弧，即它是当前森林中的一个根节点. 由于 i 是最大活跃节点，$D(i)$ 中的所有其他节点不可能是活跃节点. 重新标号后，当前树林中所有进入 i 的弧成为非允许弧，所以从当前森林中退出，即 $|D(i)|=1$. 但这并不会增加任何新的活跃节点或最大活跃节点，因此此时 Φ 最多可能增加

K 个单位.

(4) 在当前森林中增加一条新的当前(允许)弧. 此时, 不会产生任何新的最大活跃节点, 而只可能使一些节点的后代集合扩大, 因此可能使某些最大活跃节点变成非最大活跃节点. 无论如何, Φ 的值不可能增加.

我们可以把算法的执行过程分成若干个阶段来考虑, 这里每一个阶段是指算法中两次相邻的重标号操作之间的预流推进. 5.4.2 节的分析告诉我们, 预流推进算法中最多含有 $O(n^2)$ 个阶段. 我们把阶段分成两类考虑:

(1) 最多执行 $2n/K$ 次非饱和推进的阶段: 所有这类阶段中的非饱和推进的次数之和最多为 $O(n^2 * 2n/K)=O(n^3/K)$ 次.

(2) 最少执行 $2n/K$ 次非饱和推进的阶段: 由于网络中后代个数不少于 K 个的节点最多有 n/K 个, 所以此阶段至少有 n/K 次非饱和推进来自于后代个数少于 K 个的节点. 根据上面对 4 种可能情形的讨论, 这 n/K 次非饱和推进每次使得 Φ 减少至少 1 个单位. 由于算法中饱和推进的总次数为 $O(mn)$, 且所有节点重标号次数最多为 $O(n^2)$, 根据上面对 4 种可能情形的讨论, 算法中 Φ 最多增加 $O(mnK+n^2K)= O(mnK)$ 个单位. 由于开始时 Φ 最多为 nK, 算法结束时为 0, 所以所有这类阶段中的非饱和推进的次数之和最多为 $O(mnK)$ 次.

因此, 这两类非饱和推进的次数之和为 $O(n^3/K+mnK)$. 取 $K=n/m^{1/2}$, 则得到两类非饱和推进的次数之和为 $O(n^2m^{1/2})$. 因此我们得到定理 5.7.

定理 5.7　最高标号预流推进算法复杂度为 $O(n^2m^{1/2})$.　□

5.6　单位容量网络上的最大流算法

上面介绍的算法都是针对一般流网络上的最大流问题的. 在实际问题中, 网络本身往往具有一些特殊的结构, 如网络为平面图, 或者网络为二部图, 或者每条弧上的容量具有特殊的限制条件等. 如果充分利用这些网络结构的特殊性, 则可能得到针对该特殊问题的更高效(指复杂度更低)的算法. 作为一个例子, 我们这里只介绍所谓的单位容量网络. 许多 0-1 整数规划问题都可以用这种网络上的最大流模型来描述, 因此研究这种模型是很有实际价值的(在本书 7.2 节中我们将给出一个具体应用的例子).

定义 5.10　每条弧上的容量(上界)都是 1 个单位的网络称为单位容量网络(unit capacity network). 如果在这种单位容量网络中, 除了源点和汇点以外, 所有其他节点上要么最多只有一条入弧, 要么最多只有一条出弧, 则称这种网络为单位容量简单网络(unit capacity simple network).

例 5.10　如图 5.10 所示的网络是一个单位容量简单网络(假设每条弧上的容量都是 1 个单位, 因此没有在图中标出).

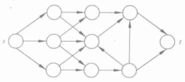


图 5.10　单位容量简单网络(例 5.10)

5.6.1　单位容量网络上的最大流算法

显然，单位容量网络上的 s-t 最大流最多为 $n-1$，因此采用 Ford-Fulkerson 标号算法求解时，复杂度为 $O(mn)$. 当采用最短增广路算法求解时，由于其瓶颈操作(增广)复杂度为 $O(mn)$，因此算法的复杂度也是 $O(mn)$. 这一复杂度都低于相应的算法在一般网络上的复杂度. 如果我们把这两种算法联合起来使用，即先采用最短增广路算法求到满足一定条件的近优解，然后从该近优解开始采用 Ford-Fulkerson 标号算法求到最优解，则可以得到总复杂度更低的算法. 具体来说，算法分成两个阶段：第 1 阶段，采用最短增广路算法求解，当源点 s 的距离标号 $d(s)> d^*=\min\{\lceil 2n^{2/3}\rceil, \lceil m^{1/2}\rceil\}$ 时停止，得到近优解 $\boldsymbol{x}'$，流值为 v'；第 2 阶段，用 Ford-Fulkerson 标号算法求到最优解 $\boldsymbol{x}^*$，流值为 v^*.

为了分析这一算法的复杂度，必须对第 1 阶段结束时的近优解 $\boldsymbol{x}'$ 的性质进行分析. 第 1 阶段结束时，我们对每一个 $k=0, 1, \cdots, d^*$，令 $V_k=\{i\in V \mid d(i)=k\}$，即 V_k 是所有距离标号等于 k 的节点集合，通常也称为残量网络中的第 k 层的节点集合. 首先假设某个 $V_k(k\neq 0)$ 为空集，记 $S=\bigcup\limits_{k+1\leqslant i\leqslant d^*} V_i$ 为所有大于等于 $k+1$ 层且小于等于 d^* 层节点的集合，则 $[S, \bar{S}]$ 为残量网络的一个最小割. 现在假设 $V_k(k\in\{1, 2, \cdots, d^*\})$ 都不为空集. 此时，根据距离标号的性质，在残量网络 $N(\boldsymbol{x}')$ 中，任何一条弧 (i, j) 一定是从某个 V_k 指向 V_{k-1}. 因此，对于任意一个 $k=1, 2, \cdots, d^*$，从 V_k 指向 V_{k-1} 的所有弧形成残量网络的一个 s-t 割. 下面对 d^* 的两种取值情况分别考虑.

假设 $d^*=\lceil 2n^{2/3}\rceil$. 此时，至少存在一个 $k\in\{1, 2, \cdots, d^*\}$，使得 $|V_k|\leqslant n^{1/3}$，$|V_{k-1}|\leqslant n^{1/3}$. 否则，如果所有相邻层的节点集合中的节点数都至少有一个大于 $n^{1/3}$，所以 $\bigcup\limits_{1\leqslant i\leqslant d^*}|V_i|> n^{1/3}d^*/2\geqslant n$，矛盾. 因此，残量网络 $N(\boldsymbol{x}')$ 中的最大流值 $v^*-v'\leqslant V_k$ 指向 V_{k-1} 的所有弧形成的割的容量 $\leqslant |V_k||V_{k-1}|\leqslant n^{2/3}\leqslant d^*$.

假设 $d^*=\lceil m^{1/2}\rceil$. 此时，对于所有 $k=1, 2, \cdots, d^*$，从 V_k 指向 V_{k-1} 的所有弧形成残量网络的 d^* 个 s-t 割. 由于这所有的 d^* 个割中弧是互不相交的，且每条弧上容量为 1，而网络中总弧数不超过 m，因此 d^*(最小割的容量)$\leqslant m$. 于是，残量网络 $N(\boldsymbol{x}')$ 中的最大流值 $v^*-v'\leqslant$ 最小割的容量 $\leqslant m/d^*\leqslant\lceil m^{1/2}\rceil=d^*$.

上面证明了 $v^*-v'\leqslant d^*$. 因此，在第 2 阶段，Ford-Fulkerson 标号算法的复杂度为

$O(d^* m)$. 那么，在第1阶段，最短增广路算法的复杂度为多少呢？在这个阶段，由于 $d(s) \leqslant d^*$，所以算法对每个节点最多执行 d^* 次重新标号操作. 由于所有弧上的容量为1，因此每次沿 (i, j) 弧的增广一定使该弧饱和，所以沿 (i, j) 弧的两次相邻增广一定使得节点 i，j 的标号都至少增加2个单位. 因此，沿每条弧 (i, j) 的(饱和)增广最多为 $d^*/2$ 次. 所以，第1阶段最短增广路算法的复杂度也是 $O(d^* m)$.

于是，我们可以得到定理5.8.

定理 5.8　用上述两阶段算法求解单位容量网络上的最大流时，复杂度为 $O(d^* m)$，即 $O(\min\{n^{2/3}m, m^{3/2}\})$. □

5.6.2　单位容量简单网络上的最大流算法

下面我们考虑一种特殊的单位容量网络，即单位容量简单网络.

仍然考虑采用上面介绍的两阶段算法，但取 $d^* = \lceil n^{1/2} \rceil$. 第1阶段结束时，记残量网络中的第 k 层的节点集合是 $V_k (k \in \{1, 2, \cdots, d^*\})$. 现在假设 V_h 是 $V_k (k \in \{1, 2, \cdots, d^*\})$ 中所含元素(节点数)最少的集合，则 $d^* |V_h| \leqslant n$，所以 $|V_h| \leqslant n/d^* \leqslant n^{1/2}$. 记 V' 为 V 中最多只有一条出弧的节点集合，令 $S = \{i | d(i)\} \cup \{i | d(i) = h \text{ 且 } i \in V'\}$. 由于 $d(s) > d^* \geqslant h$，$d(t) = 0 < h$，所以 $[S, V \backslash S]$ 是一个 s-t 割. 在残量网络的割 $[S, V \backslash S]$ 中，具有非零残留容量的弧要么指向 $V_h \cap (V \backslash V')$，要么起始点位于 $V_h \cap V'$. 所以，割 $[S, V \backslash S]$ 的残留容量至多为 $|V_h| \leqslant n^{1/2}$，即残量网络中的最大流流值 $v^* - v' \leqslant n^{1/2}$. 与5.6.1节的讨论类似，可以得到算法的总复杂度为 $O(n^{1/2} m)$. 于是，我们可以得到如下定理.

定理 5.9　用上述两阶段算法求解单位容量简单网络上的最大流时，复杂度为 $O(n^{1/2} m)$. □

练 习 题

1. 下列说法是否正确？为什么？

(1) 如果 $\boldsymbol{x}$ 是最大流，那么对于任意弧 $(i, j) \in A$，要么 $x_{ij} = 0$，要么 $x_{ji} = 0$ (假设 $(j, i) \in A$).

(2) 如果流网络中所有有向弧由相应的无向弧代替，则最大流的流值不会改变.

(3) 如果流网络中所有弧上的容量都不相同，则最小割唯一.

(4) 如果流网络中所有弧上的容量都同时乘以一个正数，则原最小割仍为最小割.

(5) 如果流网络中所有弧上的容量都同时加上一个正数，则原最小割仍为最小割.

(6) 如果流网络中所有弧上的容量都同时增加 k 个单位，则最大流的流值增加 k 的一个整数倍数.

(7) 采用最大容量增广路算法时，算法产生的增广路的容量是非增的.

(8) 如果每一个节点的距离标号是残量网络中该节点到汇点 t 的最短路的下界，则该

距离标号是有效的.

(9) 最多经过 $m+n$ 次从赢余点到源点或汇点的增广，一定可以使一个预流变换为一个最大流.

(10) 设 v^* 为最大流的流值，v' 为预流推进算法中某一阶段流入汇点 t 的流量，则一定有 $v^*-v'\leqslant\sum_{i\in I}e(i)$.（$I$ 表示所有活跃节点的集合）

2. 假设连接两城市 P,Q 的公路网如图 5.11 所示. 交通管理部门希望在公路上建立足够的车辆检查站，使得每辆从城市 P 开往城市 Q 的汽车都至少经过一个检查站. 在每条道路上建立一个检查站的费用(单位:万元)已经表示在图中，请给出一个费用最小的建站方案.

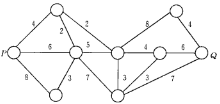


图 5.11　第 2 题图

3. 设$[S,\bar S]$和$[T,\bar T]$是流网络中的两个 s-t 割，证明 $U[S,\bar S]+U[T,\bar T]\geqslant U[S\cup T,\overline{S\cup T}]+U[S\cap T,\overline{S\cap T}]$.

4. 给出一个算法，判断一个流网络是否具有唯一的最小割，并分析你所给出的算法的计算复杂性.

5. 对于一个整数容量的流网络，假设我们已经知道它的一个非整数最大流，请你设计一个算法把它转化为一个整数最大流，并分析你所给出的算法的计算复杂性.

6. 用 Ford-Fulkerson 算法计算如图 5.12 所示网络中的最大流和最小割：

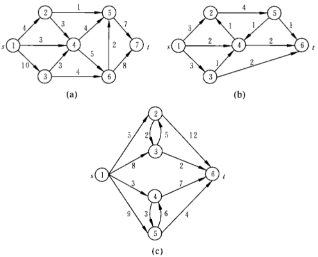


图 5.12　第 6 题图

7. 分别用最大容量增广路算法和容量变尺度算法计算练习题 6 所示网络中的最大流.

8. 用最短增广路算法计算练习题 7 所示网络中的最大流.

9. 分别用一般的预流推进算法和最高标号预流推进算法计算练习题 7 所示网络中的最大流.

10. 假设采用当前弧数据结构，证明在检索节点 i 的允许弧时，如果 $A(i)$中的一条弧在以前的迭代中不是允许弧，则在节点 i 的标号改变之前它仍然不是允许弧.

11. 通过定义势函数 $\Phi=\max\{d(i)\mid i$ 为活跃点$\}$，证明最高标号预流推进算法最多执行 $O(n^3)$次非饱和推进.

12. 网络中的最大预流是进入汇点 t 的流量尽可能大的预流.

(1) 证明：对于一个给定的最大预流 $\boldsymbol{x}^0$，存在一个最大流 $\boldsymbol{x}^*$，使得对于任意弧$(i, j)\in A$，满足 $x_{ij}^*\leqslant x_{ij}^0$.

(2) 设计一个标号算法，将一个最大预流最多经过 $m+n$ 次增广转化为一个最大流.

(3) 根据最短增广路算法的思想设计一个算法，将一个最大预流在 $O(mn)$的时间内转化为一个最大流.

(4) 根据最大标号预流推进算法的思想设计一个算法，将一个最大预流在 $O(mn)$的时间内转化为一个最大流.

13. 在一个流网络中，增加一条弧的容量可能会使得最大流的流值增加，请你设计一个算法，找出所有这样的弧. 同样，减少一条弧的容量可能会使得最大流的流值下降，请你设计一个算法，找出所有这样的弧. 分析算法的计算复杂度.

14. 用最高标号预流推进算法计算图 5.13 所示网络中的最大流(图中没有标明容量数值的弧上的容量认为是无限大). 对于给定的 n，算法执行的推进次数为多少？当参数 L，W 分别为何值时，算法执行的推进次数最多？

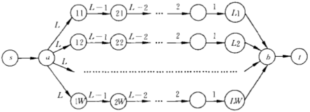


图 5.13 第 14 题图

15. 在一般的预流推进算法中，可以采用不同的规则选取活跃节点进行推进或重新标号操作. 如果分别采用以下规则，试分析算法应当采用的数据结构，相应总推进次数和运行时间.

(1) 选取距离标号最小的活跃节点.

(2) 选取赢余最大的活跃节点.

(3) 选取活跃节点，它的赢余至少是最大赢余活跃节点的赢余的一半.

(4) 选取算法最近曾经选取过的最近的活跃节点.

(5) 选取算法未选取过或曾经选取过的最远的活跃节点.

(6) 完全随机地选取活跃节点(假设产生一个随机数的时间是常数).

16. 对于流网络，如果弧$(i,j)\in A$，定义$\alpha(i,j)$是将弧(i,j)上的容量改为无穷大时最大流流值所增加的量；如果弧$(i,j)\notin A$，定义$\alpha(i,j)$是将弧(i,j)加入网络并令其上的容量为无穷大时最大流流值所增加的量. 请你设计一个算法计算所有的$\alpha(i,j)$，并分析你所给出的算法的计算复杂性.

17. 考虑弧上容量同时具有上界和下界约束的情形，即约束条件(5.2)改为

$$l_{ij}\leqslant x_{ij}\leqslant u_{ij},\quad \forall (i,j)\in A.$$

此时，应当如何求解相应的最大流问题？可行流存在的充要条件是什么？

18. 对于练习题17所考虑的网络，如何求解相应的最小流问题？能否建立相应的"最小流最大割"定理？

19. 考虑弧上和节点上的允许流量同时具有上界约束的情形，即增加约束

$$\sum_{(j,i)\in A} x_{ji}\leqslant u_i,\quad \forall i\in V.$$

此时，应当如何求解相应的最大流问题？

20. 我们知道：如果弧上的容量有无理数，则 Ford-Fulkerson 标号算法不一定在有限步内终止；并且，这一增广路算法的极限过程得到的流的流值即使收敛，也不一定收敛到最大流. 请你通过如图 5.14 所示的网络对此进行说明(所有没有标明容量的弧上的容量假定为无限). 除了图中所示的弧以外，从 1，2，3，4 中的任意两个节点及 5，6，7，8 中的任意两个节点都存在一条容量为无限的弧.

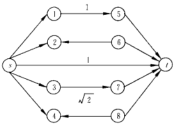


图 5.14　第 20 题图

第 6 章　最小费用流问题

我们继续考虑第 5 章开始时讨论过的例子：假设从城市 P 到城市 Q 的公路网如图 6.1 所示.假设每段公路上每天可以通过的汽车的数量有一个上限(弧上的第 1 个数，单位为万辆)，并且假设每辆汽车通过同一路段所花费的汽油费是相同的(弧上的第 2 个数，单位为百元). 如果准备每天安排 5 万辆汽车经过该公路网从城市 P 开到城市 Q，那么如何制定每辆汽车的行驶路线，可以使得所花费的汽油费总费用最小？这样的问题在网络优化中称为最小费用流问题(minimum cost flow problem).

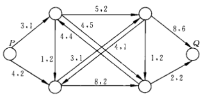


图 6.1　最小费用流问题的例子(公路网络)

6.1　最小费用流问题的数学描述

6.1.1　最小费用流问题

与第 5 章讨论过的最大流问题一样，最小费用流问题研究的对象也是网络中的流. 但是，在这一章讨论的流网络中，我们还要考虑另外一个因素，即流量流过一条弧时导致的成本(费用). 也就是说，每条弧上除了有对于流的(上下)容量限制外，还有另外一个权，即成本(费用). 这样的流网络一般称为容量-费用网络.

定义 6.1　在以 V 为节点集，A 为弧集的有向图 $G=(V,A)$ 上定义如下的权函数：

(1) C: $A\to\mathbb{R}$ 为弧上的权函数，弧 $(i,j)\in A$ 对应的权 $C(i,j)$ 记为 c_{ij}，称为弧 (i,j) 的单位流量的成本或费用(cost)；

(2) L: $A\to\mathbb{R}$ 为弧上的权函数，弧 $(i,j)\in A$ 对应的权 $L(i,j)$ 记为 l_{ij}，称为弧 (i,j) 的容量下界(lower bound)；

(3) U: $A\to\mathbb{R}$ 为弧上的权函数，弧 $(i,j)\in A$ 对应的权 $U(i,j)$ 记为 u_{ij}，称为弧 (i,j) 的容量上界，或直接称为容量(capacity)；

(4) D: $V\to\mathbb{R}$ 为节点上的权函数，节点 $i\in V$ 对应的权 $D(i)$ 记为 d_i，称为节点 i 的供需量(supply/demand)；

此时所构成的网络称为容量-费用网络，可以记为 $N=(V,A,C,L,U,D)$. □

可见，容量-费用网络也是一种流网络，所以以后我们仍然把容量-费用网络称为流网络或直接称为网络. 第 5 章关于网络流的各种概念仍然可以引入到容量-费用网络中：例如，

当 $d_i>0$ 时，表示有 d_i 个单位的外部流量流入节点 i，因此节点 i 称为供应点(supply node)或源(source)，有时也形象地称为起始点或发货点等；当 $d_i<0$ 时，表示有 d_i 个单位的流量从该节点流失到网络外部(或说被该节点吸收)，因此节点 i 称为需求点(demand node)或汇(sink)，有时也形象地称为终止点或吸收点等；当 $d_i=0$ 时，节点 i 称为转运点(transshipment node)或平衡点、中间点等.

一般来说，我们总是可以把 $L\neq 0$ 的网络转化为 $L=0$ 的网络进行研究. 所以，除非特别说明，以后我们总是假设 $L=0$ (即所有弧(i,j)的容量下界 $l_{ij}=0$)，并将 $L=0$ 时的容量-费用网络简记为 $N=(V,A,C,U,D)$.

定义 6.2 与第 5 章介绍的流的概念类似，对于容量-费用网络 $N=(V,A,C,U,D)$，其上的一个流 $\boldsymbol{x}$ 是指从 N 的弧集 A 到 $\mathbb{R}$ 的一个函数，即对每条弧$(i,j)\in A$ 赋予一个实数 x_{ij} (称为弧(i,j)的流量). 如果流 $\boldsymbol{x}$ 满足如下流量守恒条件(也称流量平衡条件)：

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = d_i, \quad \forall i \in V, \tag{6.1}$$

并且满足容量约束

$$0\leqslant x_{ij}\leqslant u_{ij}, \quad \forall (i,j)\in A, \tag{6.2}$$

则称 $\boldsymbol{x}$ 为可行流. 如果流 $\boldsymbol{x}$ 只满足容量约束条件(6.2)，但不一定满足流量守恒条件(6.1)，则称 $\boldsymbol{x}$ 为伪流(pseudoflow).

流 $\boldsymbol{x}$ 的总费用定义为

$$c(\boldsymbol{x}) = \sum_{(i,j)\in A} c_{ij}x_{ij}.$$

最小费用流问题就是在这样的网络中寻找总费用最小的可行流. 因此，用数学规划的方法，该问题可以形式地描述如下：

$$\begin{aligned}
\min\ & c(\boldsymbol{x}) = \sum_{(i,j)\in A} c_{ij}x_{ij} \\
\text{s. t.}\ & \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = d_i, \quad \forall i \in V, \\
& 0 \leqslant x_{ij} \leqslant u_{ij}, \quad \forall (i,j) \in A.
\end{aligned} \tag{6.3}$$

这种一般形式的最小费用流问题通常又称为转运问题(transshipment problem)或容量受限的转运问题(capacitated transshipment problem). 对于单源单汇的情形，即网络中只有一个起点(记为 s)和一个终点(记为 t)时，寻找从 s 流到 t 的给定流量(或最大流量、最小流量)的最小费用流，是经典的最小费用流问题. 例如，当考虑从 s 流到 t 的流量为 $v(v>0)$ 的最小费用流问题时，令 $d_s=v, d_t=-v, d_i=0$(当 $i\neq s,t$)即可.

例 6.1(续例 5.1) 在一个有向网络中，令所有弧上的容量下界为 0，容量(上界)为 1，并令图中某个节点 s 的供需量为 1，某个节点 t 的供需量为 -1，则从 s 到 t 的一条有向路正好对应于网络中的一个可行流 $\boldsymbol{x}$(当 $x_{ij}=1$ 时，表示弧(i,j)位于 s-t 路上，当$x_{ij}=0$ 时，表示弧(i,j)不在 s-t 路上). 如果再令所有弧上的(单位流量)的费用为“弧长”，则此时的最小费

用流问题就是第4章讨论过的最短路问题. 实际上,在第4章我们正是用这样的方式对最短路问题进行建模的(即式(4.1)~式(4.3)). □

可见,最短路问题是最小费用流问题的一个非常简单的特例. 正如我们在前面几章介绍的一样,约束条件(6.1)可以方便地用矩阵形式描述. 设网络的关联矩阵记为 $\boldsymbol{B}$,流向量记为 $\boldsymbol{x}$,由 d_i 组成的向量记为 $\boldsymbol{d}$,这时约束条件(6.1)即为

$$\boldsymbol{Bx}=\boldsymbol{d}. \tag{6.4}$$

根据关联矩阵的性质,我们可以立即得到如下的引理:

引理 6.1 约束条件(6.4)有解的必要条件是 $\sum_{i\in V} d_i = 0$. □

实际上,这是"流量守恒"的必然结果. 除非特别说明,本章我们总是假定这一条件是成立的.

6.1.2 最小费用流模型的特例及扩展

最小费用流问题是网络优化中的一个核心问题,许多网络优化问题都是最小费用流问题的特例. 我们从例6.1知道,最短路问题是最小费用流问题的一个特例. 前一章介绍的最大流问题也是最小费用流问题的特例. 实际上,最大流问题可以认为是如下的最小费用流问题:设 s 为起点,t 为终点,增加弧 (t,s),令 $c_{ts}=-1$, $u_{ts}=+\infty$,而令所有其他弧上的费用为0,所有节点上的供需量(外部流量)全为0.

运输问题(transportation problem)是研究比较早的最小费用流问题之一,早在1941年 Hitchcock 就进行了研究,因此运输问题又称为 Hitchcock 问题. 我们下面给出这一问题的具体描述:现有一批货物从货源地运往目的地,假设货源集合为 S,目的地集合为 T. 货源 i 可提供的货物数量为 a_i 个单位,目的地 j 对货物的需求量为 b_j 个单位,从货源 i 到目的地 j 的单位货物的运费为 c_{ij}. 如何安排运输计划使得总运费最小? 令节点集合为 $V=S\cup T$,A 为弧集,则 (V,A) 为完全二部图(当不允许从某些货源运送货物到某些目的地时,可以认为相应弧上的费用为无穷大). 因此,用数学规划的方法,该问题可以形式地描述如下:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{6.5}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} \leqslant a_i, \quad i\in S, \tag{6.6}$$

$$\sum_{i:(i,j)\in A} x_{ij} \geqslant b_j, \quad j\in T, \tag{6.7}$$

$$x_{ij}\geqslant 0, \quad (i,j)\in A. \tag{6.8}$$

如果每条弧上还有容量的限制(即约束条件为(6.3)而不是约束条件(6.8)),则称为容量受限的运输问题(capacitated transportation problem). 可以看出,运输问题的特点是只有源或汇,没有中间转运点. 显然,此问题有解的一个必要条件是

$$\sum_{i\in S} a_i \geqslant \sum_{j\in T} b_j. \tag{6.9}$$

如果$\sum_{i\in S}a_i > \sum_{j\in T}b_j$，可以增加一个虚拟的吸收点，使多余的货物量$\sum_{i\in S}a_i - \sum_{j\in T}b_j$运送到该吸收点，而相应的费用为0. 因此，可以不失一般性，认为$\sum_{i\in S}a_i = \sum_{j\in T}b_j$. 此时，约束条件(6.6)和(6.7)可以写成

$$\sum_{j:(i,j)\in A} x_{ij} = a_i, \quad i \in S, \tag{6.6a}$$

$$\sum_{i:(i,j)\in A} x_{ij} = b_j, \quad j \in T, \tag{6.7a}$$

特别地，当$a_i = b_j \equiv 1$，$|S| = |T|$时，运输问题就退化为指派问题(assigment problem)，因为此时问题的实际意义可以认为是分配$|S|$个工人(集合S)去完成$|T|$项工作(集合T)，而使总成本最小.

上面介绍的最小费用流问题所讨论的流网络有两条基本的性质：①当一定的流量流过一条弧时，该弧上导致的总费用与流量大小成线性正比关系，这样的流网络一般称为线性费用网络. 如果当一定的流量流过一条弧时，该弧上导致的总费用不一定与流量大小成线性正比关系，而是流量大小的一个凹(或凸)函数，则这样的网络称为凹(或凸)费用网络，相应的最小费用流问题称为凹(凸)费用网络上的最小费用流问题. ②当一定的流量流过一条弧时，流出该弧的流量(即流入该弧终点的流量)与进入该弧的流量(即流出该弧起点的流量)相等. 如果当一定的流量流过一条弧时，流出该弧的流量与进入该弧的流量不一定相等，而是进入该弧的流量的一个线性函数，即流出该弧的流量是对进入该弧的流量按一定比例进行放大或缩小的结果，则这样的流网络一般称为带增益(或盈亏)的流网络(flow with gain network). 可以看出，在这样的定义中，增益是发生在弧上. 同样，还可以类似地考虑增益发生在节点上的网络，从而定义节点带有增益的流网络. 更一般地，可以考虑弧和节点都带有增益的流网络.

本章中我们只讨论线性费用且不带增益的流网络，可以认为这是狭义的最小费用流模型. 而非线性费用或带增益的流网络，可以认为是广义的最小费用流模型. 我们可以用下面的图示来描述这些模型之间的联系.

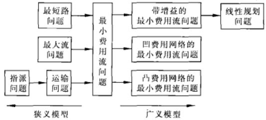


图6.2　最小费用流问题的模型

6.2　消圈算法与最小费用路算法

本节考虑传统的单源单汇网络的最小费用流问题，并用 $N=(s,t,V,A,C,U)$ 表示以 s 为起点，t 为终点的流网络，其中 V 为节点集合，A 为弧集合，C 为单位费用，U 为流量上界(流量下界假设为0). 最小费用流问题就是在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v 的最小费用流 $\boldsymbol{x}$，或者当不给定流值时，计算流值最大的最小费用流 $\boldsymbol{x}$(此时流 $\boldsymbol{x}$ 称为最小费用最大流).

6.2.1　消圈算法

对这种单源单汇的网络，我们常常把容量可行且转运点流量守恒，但流值不一定达到 v 的流称为 s-t 可行流或简称为可行流. 与讨论最大流问题时类似，我们也可以定义网络关于流的残量网络(residual network)，又常称为增量网络或辅助网络.

定义 6.3　对网络 $N=(s,t,V,A,C,U)$ 中给定的 s-t 可行流 $\boldsymbol{x}$，网络 N 关于流 $\boldsymbol{x}$ 的残量网络 $N(\boldsymbol{x})=(s,t,V,A(\boldsymbol{x}),C(\boldsymbol{x}),U(\boldsymbol{x}))$，其中 $A(\boldsymbol{x}),C(\boldsymbol{x}),U(\boldsymbol{x})$ 定义如下：

$$A(\boldsymbol{x})=\{(i,j)\mid(i,j)\in A,x_{ij}<u_{ij}\}\cup\{(i,j)\mid(j,i)\in A,x_{ji}>0\},$$

$$c_{ij}(\boldsymbol{x})=\begin{cases}c_{ij}, & (i,j)\in A,x_{ij}<u_{ij},\\ -c_{ji}, & (j,i)\in A,x_{ji}>0,\end{cases}$$

$$u_{ij}(\boldsymbol{x})=\begin{cases}u_{ij}-x_{ij}, & (i,j)\in A,x_{ij}<u_{ij},\\ x_{ji}, & (j,i)\in A,x_{ji}>0,\end{cases}$$

并称 $u_{ij}(\boldsymbol{x})$ 为弧 (i,j) 上的残留容量.

与第5章类似，我们在本章后面进行算法复杂度分析时，讨论中总是假定原网络满足这样的性质：弧 $(i,j)\in A$ 时，弧 $(j,i)\in A$，且 $c_{ji}=-c_{ij}$；并假定在残量网络中 $A(\boldsymbol{x})=A$，也就是说弧的条数和弧集合都不变. 这只要允许容量为0的弧仍然保留在网络中就可以了，因此在复杂度上不会有本质的区别.

于是，对于 $N(\boldsymbol{x})$ 中的任何一个有向圈 W，它一定对应于原网络 N 中的一个增广圈，即可以通过沿 W 对当前流 $\boldsymbol{x}$ 进行增广，获得流值相等的 s-t 可行流 $\boldsymbol{y}$. 定义 W 的费用为

$$C(W)=\sum_{(i,j)\in W}c_{ij}(\boldsymbol{x}),$$

则当增广的流量为 δ 时，$c(\boldsymbol{y})=c(\boldsymbol{x})+\delta C(W)$. 由此可见，只要 N 中存在费用为负数的增广圈 W，即 $C(W)<0$，则可以通过沿 W 对当前流 $\boldsymbol{x}$ 进行增广，获得流值相等但费用更小的 s-t 可行流 $\boldsymbol{y}$. 下面我们首先给出最小费用流的一个充要条件.

定理 6.1　设 $\boldsymbol{x}$ 为可行流，则 $\boldsymbol{x}$ 为最小费用流的充要条件是不存在负费用增广圈.

证明　由前面的讨论可以知道，必要性是显然的. 下面用反证法证明充分性. 设 $\boldsymbol{x}^0$ 为不同于 $\boldsymbol{x}$ 的可行流，但费用低于 $\boldsymbol{x}$ 的费用，即 $v(\boldsymbol{x})=v(\boldsymbol{x}^0)=v,c(\boldsymbol{x})>c(\boldsymbol{x}^0)$.

令 $\boldsymbol{x}^1=\boldsymbol{x}^0-\boldsymbol{x}$，则 $\boldsymbol{x}^1\neq\boldsymbol{0}$，$v(\boldsymbol{x}^1)=v(\boldsymbol{x}^0)-v(\boldsymbol{x})=0$，即 $\boldsymbol{x}^1$ 为网络 N 中的循环流. 根据 5.1.1 节的定理 5.1(流的分解定理)，一个循环流一定可以表示为至多 m 个非零圈流之和，所以可以将 $\boldsymbol{x}^1$ 表示为 r 个非零圈流之和($1\leqslant r\leqslant m$). 设对应的有向圈为 W_k，则

$$x_{ij}^1=\sum_{k=1}^{r}\{v(W_k)\mid(i,j)\in W_k\}.$$

于是

$$c(\boldsymbol{x}^1)=\sum_{(i,j)\in A}c_{ij}x_{ij}^1=\sum_{(i,j)\in A}c_{ij}\sum_{k=1}^{r}\{v(W_k)\mid(i,j)\in W_k\}$$
$$=\sum_{k=1}^{r}C(W_k),$$

$$c(\boldsymbol{x}^0)=c(\boldsymbol{x})+c(\boldsymbol{x}^1)=c(\boldsymbol{x})+\sum_{k=1}^{r}C(W_k).$$

因为 $c(\boldsymbol{x})>c(\boldsymbol{x}^0)$，故至少存在一个费用为负的增广圈. 矛盾，充分性得证. □

根据这一定理，可以设计如下的消圈算法(cycle-canceling algorithm)，它是由 Klein (1967)等人提出的.

消圈算法(在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v 的最小费用流 $\boldsymbol{x}$)

STEP0 在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v 的可行流 $\boldsymbol{x}$.

STEP1 在残量网络 $N(\boldsymbol{x})$ 中判别负圈. 若无负圈，则已经找到了最小费用流，结束；否则转 STEP2.

STEP2 沿找到的负圈增广流量，转 STEP1.

例 6.2 用消圈算法求如图 6.3(a)所示网络中流值为 6 的最小费用流(图中弧上的前一个数字表示弧上的容量，后一个数字表示弧上的单位费用).

首先计算流值为 6 的一个可行流 $\boldsymbol{x}_1$ 如图 6.3(b)，残量网络 $N(\boldsymbol{x}_1)$ 如图 6.3(c).

找到负费用增广圈 2-3-4-2 (费用为 $2+1-5=-2$)，增广 1 个单位的流量，得 $\boldsymbol{x}_2$ 如图 6.3(d)，残量网络 $N(\boldsymbol{x}_2)$ 如图 6.3(e).

找到负费用增广圈 2-1-3-4-2 (费用为 $-1+2+1-5=-3$)，增广 1 个单位的流量，得 $\boldsymbol{x}_3$ 如图 6.3(f)，残量网络 $N(\boldsymbol{x}_3)$ 如图 6.3(g).

此时，找不到负费用增广圈，因此 $\boldsymbol{x}_3$ 是最小费用流. □

复杂度分析：STEP0 可以采用第 5 章介绍的最大流算法，如采用最短增广路算法计算流值为 v 的可行流 $\boldsymbol{x}$ 时，其 STEP0 的复杂度为 $O(n^2m)$ (实际上这是计算最大流的复杂度，严格来说并不是计算流值为 v 的可行流的复杂度；但由于 v 的最大取值可以是最大流的流值，因此一般在分析中并不把 v 看成输入常数，而是认为其上界可以是最大流的流值).

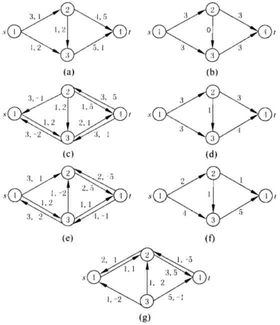


图 6.3　消圈算法的例子(例 6.2)

STEP1 在残量网络 $N(\boldsymbol{x})$ 中判别负圈可采用第 4 章介绍的最短路算法，其复杂度为 $O(nm)$. 因此算法的主要工作量在于连续寻找并消去负圈. 与第 5 章讨论复杂度时的假设一样，我们这一章也假设所有的数据都是整数，并用 C 表示所有弧上的费用中的最大者，用 U 表示所有弧上的容量以及所有节点上的供需量中的最大者（这里都指按绝对值取最大）. 由于任何可行流的费用不可能超过 mCU，而每次消去一个负圈至少使得费用下降一个单位，因此消去负圈的 STEP1 和 STEP2 最多执行 $O(mCU)$ 次. 由此可知，该算法的复杂度为 $O(nm^2CU)$，说明该算法不是多项式时间算法.

虽然消圈算法一般不是多项式时间算法，但如果按照一些特定的次序消圈，则可以得到一些多项式时间算法，这里就不介绍了（有兴趣的读者可以参看文献[1]）.

6.2.2　最小费用路算法

上面的消圈算法首先在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v 的可行流，然后通过消去负费用增广圈使费用降低. 我们能否首先在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v' 且费用最小的 s-t 可行流（$v'<v$），然后对它沿增广路增广以增加流值呢？类似于增广圈的讨论，对于 $N(\boldsymbol{x})$ 中的一条从 s 到 t 的有向路 P，它一定对应于原网络 N 中的一条增广路，即

可以通过沿 P 对当前流 $\boldsymbol{x}$ 进行增广，获得流值更大的 s-t 可行流 $\boldsymbol{y}$. 定义 P 的费用为

$$C(P)=\sum_{(i,j)\in P}c_{ij}(\boldsymbol{x}),$$

则当增广的流量为 δ 时，$c(\boldsymbol{y})=c(\boldsymbol{x})+\delta C(P)$. 为了获得最小费用流，我们希望沿费用最小的增广路对当前流 $\boldsymbol{x}$ 进行增广，以最小的费用增加获得流值更大(流值为 $v(\boldsymbol{x})+\delta$)的 s-t 可行流 $\boldsymbol{y}$. 下面我们首先给出这种增广的一个基本性质.

定理 6.2 设 $\boldsymbol{x}$ 为流值为 v 的最小费用流，P 为关于 $\boldsymbol{x}$ 的从 s 到 t 的一条最小费用增广路，且沿 P 所能增广的流量为 δ，则增广后得到流值为 $v+\delta$ 的最小费用流.

证明 记增广后得到的流为 $\boldsymbol{y}$. 显然，$\boldsymbol{y}$ 为 s-t 可行流且流值为 $v+\delta$. 由前面的充要条件，只需证明网络中不存在关于 $\boldsymbol{y}$ 的负费用增广圈. 用反证法，假设增广后存在关于 $\boldsymbol{y}$ 的负费用增广圈 W. 由于除 P 以外的弧及其流量在增广前后没有发生任何改变，于是 P 和 W 至少有一条公共弧. 不妨假设 P 和 W 只有一条公共弧，记为 (i,j)（当公共弧多于一条时可以完全类似地讨论）. 进一步分析可知，如果 (i,j) 在 P 中是正向弧，则在 W 中是反向弧；反之，如果 (i,j) 在 P 中是反向弧，则在 W 中是正向弧(因为如果不是这样，则 W 也是网络中关于流 $\boldsymbol{x}$ 的增广圈，这与 $\boldsymbol{x}$ 为最小费用流的已知条件矛盾). 于是 $P\cup W\backslash\{(i,j)\}$ 也是网络中关于 $\boldsymbol{x}$ 的增广路，且

$$C(P\cup W\backslash\{(i,j)\})=C(P)+C(W)<C(P).$$

这与 P 是关于 $\boldsymbol{x}$ 的最小费用增广路矛盾. □

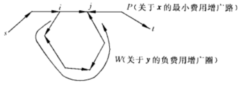


图 6.4 定理 6.2 的证明图示

根据以上讨论，可以构造如下最小费用路算法(也称为连续最短路算法，即 successive shortest path algorithm)，它是由 Jewell(1958)，Iri(1960)，Busacker & Gowen(1961)等人首先独立提出的.

最小费用路算法(在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v 的最小费用流 $\boldsymbol{x}$)

STEP0 取 $\boldsymbol{x}$ 为任一 s-t 可行流，且在同一流值的流中它是费用最小的流(如 $\boldsymbol{x}=\boldsymbol{0}$).

STEP1 若 $\boldsymbol{x}$ 的流值达到 v，计算结束；否则在残量网络 $N(\boldsymbol{x})$ 中判别最小费用路(如可用第 4 章介绍的最短路算法). 若无这样的路，则流值不可达到 v，计算结束；否则继续 STEP2.

STEP2 沿该最小费用增广路增广流量(增广后的流值不超过 v)，转 STEP1.

例6.3　用最小费用路算法求如图6.3(a)所示网络中流值为6的最小费用流.

首先从流值为0的一个可行流 $\boldsymbol{x}_0$ 开始,此时残量网络 $N(\boldsymbol{x}_0)=N$ 就是原网络,从 s 到 t 的最小费用路为1-3-4(费用为2+1=3),沿1-3-4增广4个单位流量得 $\boldsymbol{x}_1$ 如图6.5(a),残量网络 $N(\boldsymbol{x}_1)$ 如图6.5(b).

找到最小费用增广路1-2-3-4(费用为1+2+1=4),增广1个单位的流量,得 $\boldsymbol{x}_2$ 如图6.5(c),残量网络 $N(\boldsymbol{x}_2)$ 如图6.5(d).

找到最小费用增广路1-2-4(费用为1+5=6),增广1个单位的流量,得 $\boldsymbol{x}_3$ 如图6.5(e).

此时,已经找到流值为6的可行流,因此 $\boldsymbol{x}_3$ 是流值为6的最小费用流.　□

算法的复杂度分析:算法的主要工作量在于连续寻找最小费用路并增广.由于每次增广至少使得流值增加一个单位,因此STEP1和STEP2最多执行 $O(v)$ 次.由此可知,该算法的复杂度为 $O(v)$ 乘上最短路算法的复杂度.如果认为 v 是给定的参数,这当然是多项式算法.但正如前面所说,一般总是认为 v 可以取到最大流流值,因此该算法不被认为是多项式时间算法.记求解非负弧长网络的最短路算法的复杂度为 $S(n,m,C)$,则在残量网络中应用最短路算法的复杂度为 $S(n,m,C)$.又因为最大流流值不超过 nU,所以本算法复杂度为 $O(nUS(n,m,C))$.

虽然最小费用路算法一般不是多项式时间算法,但如果采用一些特定的变尺度技术,则可以得到一些多项式时间算法,这里就不介绍了(有兴趣的读者可以参看文献[1]).

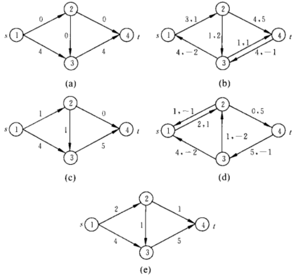


图6.5　最小费用路算法的例子(例6.3)

6.3 原始-对偶算法

6.3.1 对偶问题及互补松弛条件

本节仍然考虑传统的单源单汇网络的最小费用流问题，并用 $N=(s,t,V,A,C,U)$ 表示以 s 为起点，t 为终点的流网络，其中 V 为节点集合，A 为弧集合，C 为单位费用，U 为流量上界(流量下界假设为 0)．流值为 v 的最小费用流问题表示如下：

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{6.10}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = \begin{cases} v, & i=s, \\ -v, & i=t, \\ 0, & i\in V, i\neq s,t, \end{cases} \tag{6.11}$$

$$0 \leqslant x_{ij} \leqslant u_{ij}, \quad (i,j)\in A. \tag{6.12}$$

设网络的关联矩阵记为 $\boldsymbol{B}$，流向量记为 $\boldsymbol{x}$，约束条件(6.11)的右端向量记为 $\boldsymbol{d}$，这时约束条件(6.11)即为

$$\boldsymbol{Bx}=\boldsymbol{d}. \tag{6.13}$$

约束条件(6.12)对流量上限的约束可以等价地写成

$$-x_{ij}\geqslant -u_{ij}, \quad (i,j)\in A. \tag{6.12a}$$

对以上两类约束分别引入对偶变量 $\boldsymbol{\pi}$ 和 $\boldsymbol{z}$，则这一问题的对偶问题为

$$\max w(\boldsymbol{\pi},\boldsymbol{z}) = \sum_{i\in V} d_i\pi_i - \sum_{(i,j)\in A} u_{ij}z_{ij} \tag{6.14}$$

$$\text{s.t. } \pi_i - \pi_j - z_{ij} \leqslant c_{ij}, \quad (i,j)\in A, \tag{6.15}$$

$$z_{ij}\geqslant 0, \quad (i,j)\in A. \tag{6.16}$$

根据线性规划对偶理论，如果 $\boldsymbol{x}$ 为原问题的可行解，$\boldsymbol{\pi}$ 和 $\boldsymbol{z}$ 为对偶问题的可行解，则它们分别是所对应的问题的最优解的充要条件是它们满足以下的互补松弛条件：

$$x_{ij}(\pi_i-\pi_j-z_{ij}-c_{ij})=0, \quad (i,j)\in A, \tag{6.17}$$

$$z_{ij}(x_{ij}-u_{ij})=0, \quad (i,j)\in A. \tag{6.18}$$

下面我们说明在一定的“约定”下，这一条件可以等价地写成

$$\text{当 } \pi_i-\pi_j<c_{ij} \text{ 时}, \quad x_{ij}=0; \tag{6.19}$$

$$\text{当 } \pi_i-\pi_j>c_{ij} \text{ 时}, \quad x_{ij}=u_{ij}; \tag{6.20}$$

$$\text{当 } 0<x_{ij}<u_{ij} \text{ 时}, \quad \pi_i-\pi_j=c_{ij}. \tag{6.21}$$

实际上，利用 $\boldsymbol{x}$ 为原问题的可行解，$\boldsymbol{\pi}$ 和 $\boldsymbol{z}$ 为对偶问题的可行解，很容易从式(6.17)和式(6.18)推出式(6.19)～式(6.21)．我们所谓的“约定”，主要是针对从式(6.19)～式(6.21)推出式(6.17)和式(6.18)的．由于条件(6.19)～(6.21)没有对变量 $\boldsymbol{z}$ 的约束，因此我们约定从式(6.19)～式(6.21)推出式(6.17)和式(6.18)的意思是说存在对偶可行的变量 $\boldsymbol{z}$，使得式(6.17)和式(6.18)成立．这只需要令

$$z_{ij}=\max\{\pi_i-\pi_j-c_{ij},0\},\quad (i,j)\in A,\tag{6.22}$$

就可以了.

通常称 π_i 为节点 i 的势(potential). 由于对 π_i 没有符号(正负)限制,因此我们可以进一步得到定理 6.3.

定理 6.3 设 $\boldsymbol{x}$ 为原问题(6.10)～(6.12)的可行解,则 $\boldsymbol{x}$ 为最小费用流的充要条件是:存在节点的势 $\boldsymbol{\pi}$,满足条件(6.19)～(6.21). □

6.3.2 原始-对偶算法

与最小费用路算法的思路类似,原始-对偶算法也是从任意一个 s-t 可行流,但流值不一定达到 v 的流(如 $\boldsymbol{x}=\boldsymbol{0}$)开始迭代,直到流值达到 v 为止. 迭代过程包括两个方面:一是对弧上流的增广,一是对节点上势的修改,并且在迭代过程中始终保持最优性条件(6.19)～(6.21)成立.

为了保证在流的增广过程中始终保持最优性条件(6.19)～(6.21),原始-对偶算法只沿满足 $\pi_i-\pi_j=c_{ij}$ 的弧增广流量. 为此,通常将残量网络 $N(\boldsymbol{x})$ 中满足 $\pi_i-\pi_j=c_{ij}$ 的弧组成的子网络称为允许网络(admissible network),记为 $N^0(\boldsymbol{x})$,即

$$\begin{aligned}N^0(\boldsymbol{x})=&\{(i,j)\in A\mid \pi_i-\pi_j=c_{ij},x_{ij}<u_{ij}\}\cup\\&\{(j,i)\mid(i,j)\in A,\pi_i-\pi_j=c_{ij},x_{ij}>0\}.\end{aligned}$$

原始-对偶算法就是在允许网络 $N^0(\boldsymbol{x})$ 中寻找增广路并进行增广. 显然,$N^0(\boldsymbol{x})$ 中可以增广的最大流量就是 $N^0(\boldsymbol{x})$ 中的最大流,因此可以利用最大流算法完成.

当 $N^0(\boldsymbol{x})$ 中找不到增广路且流值小于 v 时,必须进行下一个过程,修改节点上的势. 如何修改节点上的势使得保持最优性条件(6.19)～(6.21),并且希望修改后 $N^0(\boldsymbol{x})$ 中可以找到增广路呢? 为了讨论方便,记 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$,则在 $N^0(\boldsymbol{x})$ 中的弧就是 $N(\boldsymbol{x})$ 中满足 $c_{ij}^{\pi}=0$ 的弧. 为了修改 $\boldsymbol{\pi}$ 之后 $N^0(\boldsymbol{x})$ 中出现增广路,我们希望把 $N(\boldsymbol{x})$ 中一些不满足 $c_{ij}^{\pi}=0$ 的弧也纳入 $N^0(\boldsymbol{x})$ 中. 为此,我们按照 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$ 为 (i,j) 弧的弧长在 $N(\boldsymbol{x})$ 中计算从节点 s 到所有节点 i 的最短路路长 $d(i)$. 我们后面将会证明,如果将 π_i 修改为 $\pi_i'=\pi_i-d(i)$,则最优性条件(6.19)～(6.21)仍然成立,且将会有新的弧增加到 $N^0(\boldsymbol{x})$ 中. 因此可以设计如下求解最小费用流问题的原始-对偶算法,它是由 Ford 和 Forkerson(1957,1962)首先提出的.

原始-对偶算法 (在网络 $N=(s,t,V,A,C,U)$ 中计算流值为 v 的最小费用流 $\boldsymbol{x}$)

STEP0 取 $\boldsymbol{x}$ 为任一 s-t 可行流(如 $\boldsymbol{x}=\boldsymbol{0}$),令初始势 $\boldsymbol{\pi}=\boldsymbol{0}$.

STEP1 若 $\boldsymbol{x}$ 的流值达到 v,计算结束;否则在残量网络 $N(\boldsymbol{x})$ 中,以 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$ 为 (i,j) 弧的弧长,计算从节点 s 到所有节点 i 的最短路路长 $d(i)$,并令 $\pi_i=\pi_i-d(i)$,继续 STEP2.

STEP2 在允许网络 $N^0(\boldsymbol{x})$ 中判别从 s 到 t 的最大流 (如可用第 5 章介绍的算法). 若该最大流流值为 0,则原网络中流的流值不可达到 v,计算结束;否则沿该最大流确定的增广路增广流量(增广后的流值不超过 v),转 STEP1.

为了说明算法是正确的，我们首先给出最优性条件(6.19)～(6.21)的一个等价条件.

引理 6.2 最优性条件(6.19)～(6.21)等价于对于 $N(\boldsymbol{x})$中任意的(i,j)弧，$c_{ij}^{\pi}\geqslant 0$.

证明 假设(6.19)～(6.21)成立，则对于原网络中的一条弧(i,j)，当 $c_{ij}^{\pi}<0$ 时，即 $\pi_i-\pi_j>c_{ij}$，所以根据条件(6.20)有$x_{ij}=u_{ij}$，即(i,j)弧为饱和弧，所以它不属于残量网络 $N(\boldsymbol{x})$. 也就是说，对于 $N(\boldsymbol{x})$中任意的(i,j)弧，$c_{ij}^{\pi}\geqslant 0$.

反过来，假设对于 $N(\boldsymbol{x})$中任意的(i,j)弧，$c_{ij}^{\pi}\geqslant 0$. 对于原网络中的一条弧(i,j)：

(1) 当 $\pi_i-\pi_j<c_{ij}$时，即 $c_{ij}^{\pi}>0$，所以 $c_{ji}^{\pi}=c_{ji}-\pi_j+\pi_i=-(c_{ij}-\pi_i+\pi_j)=-c_{ij}^{\pi}<0$，因此$(j,i)$弧不属于残量网络 $N(\boldsymbol{x})$. 所以 $x_{ij}=0$.

(2) 当 $\pi_i-\pi_j>c_{ij}$时，即 $c_{ij}^{\pi}<0$，因此(i,j)弧不属于残量网络 $N(\boldsymbol{x})$. 所以 $x_{ij}=u_{ij}$.

(3) 当 $0<x_{ij}<u_{ij}$时，(i,j)弧和(j,i)弧同属于残量网络 $N(\boldsymbol{x})$，因此 $c_{ij}^{\pi}\geqslant 0$ 且 $c_{ji}^{\pi}\geqslant 0$. 于是 $\pi_i-\pi_j=c_{ij}$. □

下面的引理保证算法终止时一定求到了最小费用流.

引理 6.3 在算法运行过程中，最优性条件(6.19)～(6.21)成立.

证明 算法开始时，最优性条件(6.19)～(6.21)显然成立. 因此只需要证明，如果 π_i 满足式(6.19)～式(6.21)，则将 π_i 修改为$\pi_i'=\pi_i-d(i)$后，π_i' 仍然满足式(6.19)～式(6.21).

根据上面的引理 6.2，对于 $N(\boldsymbol{x})$中任意的(i,j)弧，一定有$c_{ij}^{\pi}\geqslant 0$，且我们只需要证明 $c_{ij}^{\pi'}\geqslant 0$. 根据最短路方程，$d(j)-d(i)\leqslant c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$，即 $c_{ij}-(\pi_i-d(i))+(\pi_j-d(j))\geqslant 0$. 于是 $c_{ij}-\pi_i'+\pi_j'\geqslant 0$，即 $c_{ij}^{\pi'}\geqslant 0$. □

最后，我们还需要证明，如果将 π_i 修改为$\pi_i'=\pi_i-d(i)$，则将会有新的弧增加到 $N^0(\boldsymbol{x})$中. 这可以从下面的引理得到.

引理 6.4 对于 $N(\boldsymbol{x})$中一条弧(i,j)，如果它位于从起点 s 到某一节点的最短路上(以c_{pq}^{π}为(p,q)弧的弧长)，则 $c_{ij}^{\pi'}=0$.

证明 对于最短路上的弧(i,j)，一定有 $d(j)-d(i)=c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$，即 $c_{ij}^{\pi'}=0$. □

实际上，这一引理说明，只要 $N(\boldsymbol{x})$中从 s 到 t 存在有向路(增广路)，则 π_i 修改为 π_i'后，$N^0(\boldsymbol{x})$中从 s 到 t 存在有向路(增广路)即可以对当前流进行增广.

上面几个引理和讨论说明，前面介绍的算法是正确的. 这一算法的特点是，在运行中始终保持对偶变量$\boldsymbol{\pi}$为对偶问题的可行解，并始终保持互补松弛条件，但原始变量 $\boldsymbol{x}$ 在算法终止前通常不是原问题的可行解(即只是伪流，而不是流值为 v 的可行流). 因此，这一算法称为原始-对偶算法. 比较该算法与 6.2 节介绍的最小费用路算法，虽然最小费用路算法中我们没有显式地引入对偶变量$\boldsymbol{\pi}$，但我们也可以与本节介绍的算法一样引入对偶变量$\boldsymbol{\pi}$，因此可以认为最小费用路算法也是保持对偶可行和互补松弛条件的，是一种特殊的原始-对偶算法，即要求增广流量时每次沿"一条"最小费用路增广. 本节介绍的原始-对偶算法实际上也是沿最小费用路增广流值，只是可能每次沿"多条"最小费用路同时增广(即最小费用路最大流增广). 这可以从下面的引理看出.

引理 6.5 对于 $N(\boldsymbol{x})$中从任一节点 p 到另一节点 q 的一条有向路 P，

$$\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(p) + \pi(q).$$ □

直接应用 c_{ij}^{π} 的定义就可以得到上述结果. 根据这一结果，对于 $N(\boldsymbol{x})$ 中从 s 到 t 的一条有向路(增广路)P，一定有 $\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(s) + \pi(t)$，即

$$C(P) = \sum_{(i,j)\in P} c_{ij} = \sum_{(i,j)\in P} c_{ij}^{\pi} + \pi(s) - \pi(t).$$

对于给定的对偶变量$\boldsymbol{\pi}$，由于 $c_{ij}^{\pi} \geqslant 0$，所以满足 $c_{ij}^{\pi}=0$ 的弧(即允许网络 $N^0(\boldsymbol{x})$中的弧)组成的增广路是 $N(\boldsymbol{x})$中的最小费用路. 所以，本节介绍的原始-对偶算法实际上也是沿最小费用路增广流值，只是每次可能会沿"多条"最小费用路同时增广(即最小费用路最大流增广).

例 6.4　用原始-对偶算法计算如图 6.6(a)网络中的最小费用流(图中弧上的前一个数字表示弧上的容量，后一个数字表示弧上的单位费用；节点上的数字表示节点的供需量).

这一问题是一个最小费用循环流问题，并且显然可以直观地看出只有唯一的可行流. 我们下面根据算法的具体步骤进行计算：首先引进两个新节点(源 s 和汇 t)，可以把问题转化为计算单源单汇的流值为 4 的最小费用流问题. 即网络 N 如图 6.6(b).

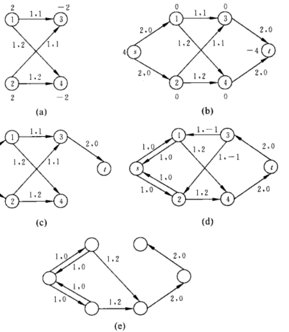


图 6.6　原始-对偶算法计算最小费用流的例子(例 6.4)

令初始流 $\boldsymbol{x}=\boldsymbol{0}$，初始势$\boldsymbol{\pi}=\boldsymbol{0}$，此时残量网络 $N(\boldsymbol{x})=N$ 就是上面的网络. 以 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j=c_{ij}$ 为弧长，计算从 s 到各节点的最短路路长为：$d(s)=d(1)=d(2)=0, d(3)=1,$

$d(4)=2, d(t)=1$. 写成向量形式，即 $\boldsymbol{d}=(0,0,0,1,2,1)$. 修改$\boldsymbol{\pi}$得到 $\boldsymbol{\pi}=(0,0,0,-1,-2,-1)$，此时允许网络 $N^0(\boldsymbol{x})$ 如图 6.6(c).

在此时网络 $N^0(\boldsymbol{x})$中，计算从 s 到 t 的最大流流值为 2（沿 s-1-3-t 和 s-2-3-t 各发送一个单位的流量）. 此时残量网络 $N(\boldsymbol{x})$ 如图 6.6(d).

以 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$ 为弧长，计算从 s 到各节点的最短路路长为 $\boldsymbol{d}=(0,0,0,1,0,1)$. 修改$\boldsymbol{\pi}$ 得到$\boldsymbol{\pi}=(0,0,0,-2,-2,-2)$，此时允许网络 $N^0(\boldsymbol{x})$如图 6.6(e).

在此时网络 $N^0(\boldsymbol{x})$中，计算从 s 到 t 的最大流流值为 2（沿 s-1-4-t 和 s-2-4-t 各发送一个单位的流量）. 此时各节点流量达到平衡，即已经得到最小费用流为 $x_{13}=x_{14}=x_{23}=x_{24}=1$. □

算法复杂度分析：从算法的过程可以看出，每次循环迭代修改弧上的流值和节点上的势各一次. 由于流值不可能超过 nU，且任何节点上的势不可能低于 $-nC$，因此总迭代次数不会超过 $\min\{nU,nC\}$. 记求解非负弧长网络的最短路算法的复杂度为 $S(n,m,C)$，最大流算法的复杂度为 $M(n,m,U)$，则本算法复杂度为 $O(\min\{nU,nC\}[S(n,m,nC)+M(n,m,U)])$. 这一算法仍然不是多项式时间算法，但与最小费用路算法相比，可能会以每次迭代调用一次最大流算法为代价，希望减少一些迭代次数.

6.4 瑕疵算法

前面介绍的最小费用路算法和原始-对偶算法的特点是，在运行中始终保持对偶变量$\boldsymbol{\pi}$为对偶问题的可行解，并始终保持互补松弛条件，但原始变量 $\boldsymbol{x}$ 在算法终止前通常不是原问题的可行解(即只是伪流，而不一定满足节点上的流量守恒条件，即不是流值为 v 的可行流). 本节介绍的瑕疵算法(out-of-kilter algorithm，有些中文书中也翻译为“状态算法”)在迭代过程中则对这些条件更为放宽，即只要求满足节点上的流量守恒条件，而不要求 $\boldsymbol{x}$ 为伪流(即可能不满足容量约束)，并且也不一定保持互补松弛条件. 然而，算法的思想是通过一定的步骤，使得这些非最优的“程度”不断降低，最后达到最优. 可以认为，瑕疵算法是原始-对偶算法的一种变形.

首先回顾一下 6.3 节介绍的互补松弛条件(6.19)～(6.21)，并借用 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$ 的记号重新叙述如下：

$$\text{当 } c_{ij}^{\pi}>0 \text{ 时}, \quad x_{ij}=0; \tag{6.19a}$$

$$\text{当 } c_{ij}^{\pi}<0 \text{ 时}, \quad x_{ij}=u_{ij}; \tag{6.20a}$$

$$\text{当 } 0<x_{ij}<u_{ij} \text{ 时}, \quad c_{ij}^{\pi}=0. \tag{6.21a}$$

前面几节中讨论的算法都假定弧上的下容量为 0，上面这一互补松弛条件也是针对弧上的下容量为 0 而建立的. 在瑕疵算法的迭代过程中，算法并不要求流 $\boldsymbol{x}$ 满足容量约束，因此我们可以考虑弧上的下容量不为 0 的一般情况. 此外，在瑕疵算法的迭代过程中，流 $\boldsymbol{x}$ 必须满足节点上的守恒条件，因此我们这里假设流 $\boldsymbol{x}$ 为循环流(circulation)，即流值为 0 的可行流（想一想，非循环流的情况是否都可以转化为循环流的情况）. 对于循环流，自然没有所

谓源点和汇点，网络中的所有节点都是转运点. 此时模型可以重新描述如下：

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0, \quad i \in V,$$

$$l_{ij} \leqslant x_{ij} \leqslant u_{ij}, \quad (i,j) \in A.$$

类似于6.3节推导条件(6.19)～(6.21)的过程，可以得到(留作练习)如下关于式(6.23)～式(6.25)的互补松弛条件(仍然定义 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$)：

$$\text{当 } c_{ij}^{\pi} > 0 \text{ 时}, \qquad x_{ij} = l_{ij}; \tag{6.23}$$

$$\text{当 } c_{ij}^{\pi} < 0 \text{ 时}, \qquad x_{ij} = u_{ij}; \tag{6.24}$$

$$\text{当 } l_{ij} < x_{ij} < u_{ij} \text{ 时}, \quad c_{ij}^{\pi} = 0. \tag{6.25}$$

在瑕疵算法中，我们把上述条件称为 Kilter 条件(或翻译为瑕疵条件、状态条件等)，并称满足上述条件的弧为无瑕的(in-kilter)，否则称为有瑕的(out-of-kilter). 对于每条弧 (i,j)，其对应的 Kilter 条件可以用如图6.7所示的 Kilter 图(或翻译为瑕疵图、状态图)表示，只有粗实线上的点 (x_{ij}, c_{ij}^{π}) 是满足 Kilter 条件的. 如果所有弧都是无瑕的，则得到了原问题的最优解.

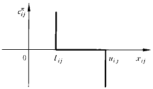


图6.7 Kilter 图

定义 6.4 对于每条弧 (i,j)，我们定义其 Kilter 数(或翻译为瑕疵数、状态数)为将流量 x_{ij} 修正为满足 Kilter 条件所需要修改的量(假设保持节点上的势不变)，记为 $K(x_{ij})$ 或 k_{ij}，即

$$K(x_{ij})=k_{ij}=\begin{cases} |x_{ij}-l_{ij}|, & \text{当 } c_{ij}^{\pi}>0 \text{ 时}, \\ |x_{ij}-u_{ij}|, & \text{当 } c_{ij}^{\pi}<0 \text{ 时}, \\ l_{ij}-x_{ij}, & \text{当 } c_{ij}^{\pi}=0 \text{ 且 } x_{ij}<l_{ij} \text{ 时}, \\ x_{ij}-u_{ij}, & \text{当 } c_{ij}^{\pi}=0 \text{ 且 } x_{ij}>u_{ij} \text{ 时}, \\ 0, & \text{其他}. \end{cases}$$

瑕疵算法的思想就是要在算法过程中使弧上的 Kilter 数不断下降，最后降为0. 瑕疵算法仍然是在残量网络上对弧上的流量进行操作，但由于算法并不要求在执行过程中的流一定满足容量约束，因此以前我们定义的残量网络的构造方法不一定够用，所以我们下面首先给出这时残量网络的构造方法.

从前面的讨论中我们知道，构造残量网络的基本想法是：把原网络中可能增加流量的弧(前向弧)或可能减少流量的弧(反向弧)纳入残量网络. 因此，对于一个可能不满足容量约束的流 $\boldsymbol{x}$，其残量网络 $N(\boldsymbol{x})$ 的构造方法如下：

(1) 对于满足容量约束的弧，即当 $l_{ij} \leqslant x_{ij} \leqslant u_{ij}$ 时，其处理与前面章节中的讨论是一样的. 如果 $x_{ij} < u_{ij}$，则弧 (i,j) 属于残量网络，残留容量为 $u_{ij}-x_{ij}$，费用为 c_{ij}；如果 $x_{ij} > l_{ij}$，则弧 (j,i) 属于残量网络，残留容量为 $x_{ij}-l_{ij}$，费用为 $-c_{ij}$.

(2) 对于不满足下容量约束的弧，即当 $x_{ij}<l_{ij}$ 时，为了降低不可行的程度，弧上的流量应当增加. 此时弧(i,j)属于残量网络，残留容量为 $l_{ij}-x_{ij}$，费用为 c_{ij}.

(3) 对于不满足上容量约束的弧，即当 $x_{ij}>u_{ij}$ 时，为了降低不可行的程度，弧上的流量应当减少. 此时弧(j,i)属于残量网络，残留容量为 $x_{ij}-u_{ij}$，费用为$-c_{ij}$.

从这一构造过程可以知道，原网络中的任何一条瑕疵弧一定会在残量网络中得到反映(即瑕疵弧不是以前向弧的形式出现在残量网络中，就会以反向弧的形式出现). 为了叙述的方便，我们下面有时直接说残量网络中的某条弧是瑕疵弧，意思是指它对应的原网络中的那条弧为瑕疵弧. 更进一步，根据原网络中弧上的 Kilter 数的定义，我们可以直接在残量网络中定义弧上的 Kilter 数，分 3 种情况讨论如下：

(1) 残量网络中的弧(i,j)在原网络中对应的弧上的流量满足容量约束. 如果 $c_{ij}^{\pi}\geqslant 0$，那么令其 Kilter 数为 0，即(i,j)是无瑕弧；如果 $c_{ij}^{\pi}<0$，那么令其 Kilter 数为残留容量，即(i,j)是瑕疵弧.

(2) 残量网络中的弧(i,j)在原网络中对应的弧上的流量不满足下容量约束($x_{ij}<l_{ij}$). 此时 (i,j)是瑕疵弧，且当 $c_{ij}^{\pi}\geqslant 0$ 时其 Kilter 数等于(i,j)的残留容量 $l_{ij}-x_{ij}$；当 $c_{ij}^{\pi}<0$ 时其 Kilter 数等于 $u_{ij}-x_{ij}$.

(3) 残量网络中的弧(i,j)在原网络中对应的弧上的流量不满足上容量约束($x_{ji}>u_{ji}$). 此时(i,j)是瑕疵弧，且当 $c_{ji}^{\pi}>0$(即$c_{ij}^{\pi}<0$)时其 Kilter 数等于 $x_{ji}-l_{ji}$；当 $c_{ji}^{\pi}\leqslant 0$(即 $c_{ij}^{\pi}\geqslant 0$)时其 Kilter 数等于 $x_{ji}-u_{ji}$.

瑕疵算法是由 Yakovleva(1959)，Minty(1960)，Fulkerson (1961)等人分别独立提出的，Aashtiani 和 Magnanti(1976)给出了该算法的一种高效率的实现方法. 该算法可描述如下：

瑕疵算法(计算最小费用循环流 $\boldsymbol{x}$)

STEP0　给出初始势和初始循环流(如$\boldsymbol{\pi}=\boldsymbol{0},\boldsymbol{x}=\boldsymbol{0}$).

STEP1　计算弧上的 Kilter 数. 若网络中不存在瑕疵弧，则已经得到最优解，计算结束；否则在残量网络 $N(\boldsymbol{x})$中，选择一条瑕疵弧(p,q)，继续下一步.

STEP2　在 $N(\boldsymbol{x})\backslash\{(q,p)\}$中，以 $\max\{0,c_{ij}^{\pi}\}$为(i,j)弧的弧长，计算从节点 q 到所有节点 i 的最短路路长 $d(i)$，并记从节点 q 到节点 p 的最短路为 P. 令 $\pi_i=\pi_i-d(i)$，继续 STEP3.

STEP3　若(p,q)仍为瑕疵弧，则沿 $P\cup\{(p,q)\}$确定的增广圈增广流量(增广的流值为该增广圈的容量)，转 STEP1；否则直接转 STEP1.

上面的算法主要包括两个过程：一是对节点上势的修改，一是沿增广圈增广流量. 下面两个引理可以说明此算法是正确的最小费用循环流算法.

引理 6.6　在瑕疵算法中，对节点上势的修改不会增加任何弧上的 Kilter 数.

证明　对残量网络 $N(\boldsymbol{x})$ 的一条弧 $(i,j)\neq(p,q)$，由最短路方程得 $d(j)\leqslant d(i)+\max\{0,c_{ij}^{\pi}\}$，即 $d(i)-d(j)\geqslant-\max\{0,c_{ij}^{\pi}\}$. 记修改后的势 $\pi_i'=\pi_i-d(i)$，则

$$\begin{aligned}c_{ij}^{\pi'}&=c_{ij}-\pi_i'+\pi_j'=c_{ij}-(\pi_i-d(i))+(\pi_j-d(j))\\&=c_{ij}^{\pi}+(d(i)-d(j))\geqslant c_{ij}^{\pi}-\max\{0,c_{ij}^{\pi}\}.\end{aligned}$$

所以，当 $c_{ij}^{\pi}\geqslant0$ 时，

$$c_{ij}^{\pi'}\geqslant c_{ij}^{\pi}-\max\{0,c_{ij}^{\pi}\}=0;\tag{6.26}$$

当 $c_{ij}^{\pi}<0$ 时，

$$c_{ij}^{\pi'}\geqslant c_{ij}^{\pi}-\max\{0,c_{ij}^{\pi}\}=c_{ij}^{\pi}.\tag{6.27}$$

而对于弧 $(i,j)=(p,q)$，由于 $d(q)=0,d(p)>0$，所以

$$\begin{aligned}c_{ij}^{\pi'}&=c_{ij}-\pi_i'+\pi_j'=c_{ij}-(\pi_i-d(i))+(\pi_j-d(j))\\&=c_{ij}^{\pi}+d(i)\geqslant c_{ij}^{\pi}.\end{aligned}\tag{6.28}$$

从 Kilter 数的定义我们可以知道：当弧上流量保持不变时，只有 c_{ij}^{π} 的符号改变时，弧上的 Kilter 数才可能改变. 下面分别对几种情况讨论：

(1) 残量网络中的弧 (i,j) 在原网络中对应的弧上的流量满足容量约束. 此时，如果 (i,j) 关于 π 是无瑕弧(其 Kilter 数为 0)，则 $c_{ij}^{\pi}\geqslant0$，由式(6.26)知 $c_{ij}^{\pi'}\geqslant0$，所以 (i,j) 关于 π' 仍然是无瑕弧. 此外，如果 (i,j) 关于 π 是瑕疵弧(其 Kilter 数必然等于 (i,j) 的残留容量)，则 $c_{ij}^{\pi}<0$，由式(6.27)和式(6.28)知 $c_{ij}^{\pi'}\geqslant c_{ij}^{\pi}$，所以 (i,j) 关于 π' 要么是无瑕弧(即 $c_{ij}^{\pi'}\geqslant0$)，要么仍然是瑕疵弧(即 $c_{ij}^{\pi'}<0$，此时其 Kilter 数在节点上势的修改前后都等于 (i,j) 的残留容量). 无论哪种情况，节点上势的修改都不会增加任何弧上的 Kilter 数.

(2) 残量网络中的弧 (i,j) 在原网络中对应的弧 (i,j) 上的流量不满足下容量约束 $(x_{ij}<l_{ij})$. 此时在节点上势的修改前后 (i,j) 一定都是瑕疵弧. 当 $c_{ij}^{\pi}\geqslant0$ 时，由式(6.26)和式(6.28)知 $c_{ij}^{\pi'}\geqslant0$，其 Kilter 数在节点上势的修改前后都等于 (i,j) 的残留容量 $l_{ij}-x_{ij}$. 当 $c_{ij}^{\pi}<0$ 时，由式(6.27)和式(6.28)知 $c_{ij}^{\pi'}\geqslant c_{ij}^{\pi}$. 若 $c_{ij}^{\pi'}<0$，则其 Kilter 数在节点上势的修改前后都等于 $u_{ij}-x_{ij}$；若 $c_{ij}^{\pi'}\geqslant0$，则 (i,j) 关于 π' 的 Kilter 数等于 $l_{ij}-x_{ij}\leqslant u_{ij}-x_{ij}$. 无论哪种情况，节点上势的修改都不会增加任何弧上的 Kilter 数.

(3) 残量网络中的弧 (i,j) 在原网络中对应的弧 (j,i) 上的流量不满足上容量约束 $(x_{ji}>u_{ji})$. 此时在节点上势的修改前后 (i,j) 一定都是瑕疵弧. 在节点上势的修改之前，当 $c_{ji}^{\pi}>0$ 时其 Kilter 数等于 $x_{ji}-l_{ji}$；当 $c_{ji}^{\pi}\leqslant0$ 时其 Kilter 数等于 $x_{ji}-u_{ji}\leqslant x_{ji}-l_{ji}$. 或者等价地说，当 $c_{ij}^{\pi}=-c_{ji}^{\pi}<0$ 时其 Kilter 数等于 $x_{ji}-l_{ji}$；当 $c_{ij}^{\pi}=-c_{ji}^{\pi}\geqslant0$ 时其 Kilter 数等于 $x_{ji}-u_{ji}\leqslant x_{ji}-l_{ji}$. 与情况(2)类似讨论可知，节点上势的修改都不会增加任何弧上的 Kilter 数.

综上所述，引理成立.　□

引理 6.7　在瑕疵算法中，沿增广圈 $P\cup\{(p,q)\}$ 增广流量不会增加任何弧上的 Kilter 数，且弧 (p,q) 的 Kilter 数严格减少.

证明　沿增广圈 $P\cup\{(p,q)\}$ 增广流量只可能改变圈上的弧及其反向弧的 Kilter 数. 显然，流量增广不改变容量可行弧的容量可行性. 对于增广圈上容量不可行(即流量不满足容量约束)的弧 (i,j)，则流量增广后该弧的 Kilter 数一定严格下降，且由残量网络的构造方

法可知,不会在残量网络中加入新的弧(j,i).所以下面只考虑增广圈上的容量可行的弧.

对于最短路P上的容量可行弧(i,j),由最短路方程有$d(j)=d(i)+\max\{0,c_{ij}^{\pi}\}\geqslant d(i)+c_{ij}^{\pi}$,所以

$$\begin{aligned}c_{ij}^{\pi'} &= c_{ij}-\pi_i'+\pi_j' = c_{ij}-(\pi_i-d(i))+(\pi_j-d(j))\\ &= c_{ij}^{\pi}+(d(i)-d(j))\leqslant 0.\end{aligned}$$

因此,若弧(i,j)在原网络中对应的弧为(i,j),则流的增广不会增加其 Kilter 数,且当$c_{ij}^{\pi'}<0$时实际上会严格减少其 Kilter 数.若弧(i,j)在原网络中对应的弧为(j,i),则由$c_{ji}^{\pi'}=-c_{ij}^{\pi'}\geqslant 0$可知,流的增广也不会增加其 Kilter 数,且当$c_{ij}^{\pi'}<0$时实际上会严格减少其 Kilter 数.

注意到沿容量可行弧(i,j)的增广可能会在残量网络中加入新的弧(j,i).但由于$c_{ji}^{\pi'}=-c_{ij}^{\pi'}\geqslant 0$,所以在残量网络中新加入的弧$(j,i)$是满足 Kilter 条件的.

最后看看弧(p,q),也只需要分析它为容量可行的弧的情况.由算法 STEP3,只有它为瑕疵弧时才进行增广,即$c_{pq}^{\pi'}<0$,因此增广会严格减少其 Kilter 数.此外,由于$c_{qp}^{\pi'}=-c_{pq}^{\pi'}>0$,因此增广也不会使$(q,p)$成为残量网络中的瑕疵弧.

综上所述,引理得证. □

例 6.5 用瑕疵算法计算如图 6.8(a)所示网络中的最小费用循环流(图中弧上的前两个数字分别表示弧上的容量下界和上界,第 3 个数字表示弧上的单位费用).

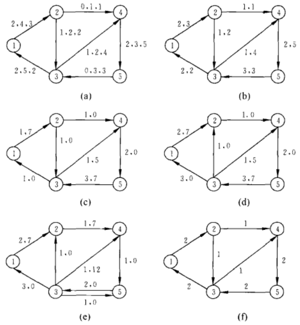


图 6.8 瑕疵算法计算最小费用流的例子(例 6.5)

从初始势$\pi=\mathbf{0}$和初始循环流$x=\mathbf{0}$开始计算. 计算弧上的各种状态量和 Kilter 数：

弧	l_{ij}	u_{ij}	x_{ij}	c_{ij}	c_{ij}^{π}	k_{ij}
(1,2)	2	4	0	3	3	2
(2,3)	1	2	0	2	2	1
(2,4)	0	1	0	1	1	0
(3,1)	2	5	0	2	2	2
(3,4)	1	2	0	4	4	1
(4,5)	2	3	0	5	5	2
(5,3)	0	3	0	3	3	0

此时网络中存在瑕疵弧. 构造残量网络$N(x)$如图 6.8(b)所示(图中弧上的第 1 个数字表示弧上的残留容量,第 2 个数字表示弧上的变换后的费用c_{ij}^{π}).

在残量网络$N(x)$中,选择一条瑕疵弧$(p,q)=(1,2)$. 在$N(x)\backslash\{(2,1)\}=N(x)$中,以$\max\{0,c_{ij}^{\pi}\}=c_{ij}$为$(i,j)$弧的弧长,计算从节点$q=2$到所有节点$i$的最短路路长$d(1)=4$, $d(2)=0,d(3)=2,d(4)=1,d(5)=6$,即$\boldsymbol{d}=(4,0,2,1,6)$. 从节点$q=2$到节点$p=1$的最短路$P$为 2-3-1. 令$\pi_i=\pi_i-d(i)$,得$\boldsymbol{\pi}$为$(-4,0,-2,-1,-6)$. 易知$(p,q)=(1,2)$仍为瑕疵弧,所以沿$P\cup\{(p,q)\}=$2-3-1-2 确定的增广圈增广流量(增广的流值为该增广圈的容量 1). 此时计算弧上的各种状态量和 Kilter 数得：

弧	l_{ij}	u_{ij}	x_{ij}	c_{ij}	c_{ij}^{π}	k_{ij}
(1,2)	2	4	1	3	7	1
(2,3)	1	2	1	2	0	0
(2,4)	0	1	0	1	0	0
(3,1)	2	5	1	2	0	1
(3,4)	1	2	0	4	5	1
(4,5)	2	3	0	5	0	2
(5,3)	0	3	0	3	7	0

网络中仍然存在瑕疵弧. 构造残量网络$N(x)$如图 6.8(c)所示. 在残量网络$N(x)$中,仍然选择瑕疵弧$(p,q)=(1,2)$. 在$N(x)\backslash\{(2,1)\}=N(x)$中,以$\max\{0,c_{ij}^{\pi}\}=c_{ij}$为$(i,j)$弧的弧长,计算从节点$q=2$到所有节点$i$的最短路路长,即$\boldsymbol{d}=(0,0,0,0,0)$. 从节点$q=2$到节点$p=1$的最短路$P$仍为 2-3-1. 令$\pi_i=\pi_i-d(i)$,得$\boldsymbol{\pi}=(-4,0,-2,-1,-6)$. 易知$(p,q)=(1,2)$仍为瑕疵弧,所以沿$P\cup\{(p,q)\}=$2-3-1-2 确定的增广圈增广流量(增广的流值为该增广圈的容量 1). 此时计算弧上的各种状态量和 Kilter 数得：

弧	l_{ij}	u_{ij}	x_{ij}	c_{ij}	c_{ij}^{π}	k_{ij}
(1,2)	2	4	2	3	7	0
(2,3)	1	2	2	2	0	0
(2,4)	0	1	0	1	0	0
(3,1)	2	5	2	2	0	0
(3,4)	1	2	0	4	5	1
(4,5)	2	3	0	5	0	2
(5,3)	0	3	0	3	7	0

网络中仍然存在瑕疵弧.构造残量网络 $N(\boldsymbol{x})$ 如图 6.8(d)所示.在残量网络 $N(\boldsymbol{x})$ 中,选择一条瑕疵弧 $(p,q)=(3,4)$.在 $N(\boldsymbol{x})\setminus\{(4,3)\}=N(\boldsymbol{x})$ 中,以 $\max\{0,c_{ij}^{\pi}\}=c_{ij}$ 为 (i,j) 弧的弧长,计算从节点 $q=4$ 到所有节点 i 的最短路路长,即 $\boldsymbol{d}=(7,7,7,0,0)$.从节点 $q=4$ 到节点 $p=3$ 的最短路 P 为 4-5-3.令 $\pi_i=\pi_i-d(i)$,得 $\boldsymbol{\pi}=(-11,-7,-9,-1,-6)$.易知 $(p,q)=(3,4)$ 仍为瑕疵弧,所以沿 $P\cup\{(p,q)\}=$4-5-3-4 确定的增广圈增广流量(增广的流值为该增广圈的容量 1).此时计算弧上的各种状态量和 Kilter 数得:

弧	l_{ij}	u_{ij}	x_{ij}	c_{ij}	c_{ij}^{π}	k_{ij}
(1,2)	2	4	2	3	7	0
(2,3)	1	2	2	2	0	0
(2,4)	0	1	0	1	7	0
(3,1)	2	5	2	2	0	0
(3,4)	1	2	1	4	12	0
(4,5)	2	3	1	5	0	1
(5,3)	0	3	1	3	0	0

网络中仍然存在瑕疵弧.构造残量网络 $N(\boldsymbol{x})$ 如图 6.8(e)所示.在残量网络 $N(\boldsymbol{x})$ 中,选择一条瑕疵弧 $(p,q)=(4,5)$.在 $N(\boldsymbol{x})\setminus\{(5,4)\}=N(\boldsymbol{x})$ 中,以 $\max\{0,c_{ij}^{\pi}\}=c_{ij}$ 为 (i,j) 弧的弧长,计算从节点 $q=5$ 到所有节点 i 的最短路路长,即 $\boldsymbol{d}=(0,0,0,7,0)$.从节点 $q=5$ 到节点 $p=4$ 的最短路 P 为 5-3-2-4.令 $\pi_i=\pi_i-d(i)$,得 $\boldsymbol{\pi}=(-11,-7,-9,-8,-6)$.易知 $(p,q)=(4,5)$ 仍为瑕疵弧,所以沿 $P\cup\{(p,q)\}=$5-3-2-4-5 确定的增广圈增广流量(增广的流值为该增广圈的容量 1).此时计算弧上的各种状态量和 Kilter 数得:

弧	l_{ij}	u_{ij}	x_{ij}	c_{ij}	c_{ij}^{π}	k_{ij}
(1,2)	2	4	2	3	7	0
(2,3)	1	2	1	2	0	0
(2,4)	0	1	1	1	0	0
(3,1)	2	5	2	2	0	0
(3,4)	1	2	1	4	5	0
(4,5)	2	3	2	5	7	0
(5,3)	0	3	2	3	0	0

此时网络中不存在瑕疵弧，因此已经得到最小费用循环流．最小费用循环流如图6.8(f)．

□

算法复杂度分析：从算法的过程可以看出，每次循环迭代修改圈上的流值和节点上的势各一次，并使所有弧上的总Kilter数至少下降1个单位．设初始流和势对应的总Kilter数为K，则总迭代次数不会超过K．记求解非负弧长网络的最短路算法的复杂度为$S(n,m,C)$，则本算法复杂度为$O(K\,S(n,m,nC))$．特别地，当取初始流$\boldsymbol{x}=\boldsymbol{0}$时，由于每条弧上的Kilter数不超过$U$，算法复杂度可以写成为$O(mU\,S(n,m,nC))$．由此可以看出，瑕疵算法仍然不是多项式时间算法．

6.5 松弛算法

对于具有复杂约束的优化问题，特别是整数规划问题，Lagrange松弛方法是一种常用的技巧．在最小费用流问题的线性规划模型中，流量守恒方程(6.1)是一个非常基本的约束，如果对该约束进行松弛，结果会如何呢？

对约束(6.1)引入Lagrange乘子π_i(仍然称为i节点上的势，由于约束(6.1)为等式，所以π_i没有符号限制)，则得到如下的Lagrange松弛问题：

$$LR(\boldsymbol{\pi}) = \min c(\boldsymbol{x},\boldsymbol{\pi})$$

$$= \sum_{(i,j)\in A} c_{ij}x_{ij} + \sum_{i\in V}\pi_i\Big(d_i - \sum_{j:(i,j)\in A} x_{ij} + \sum_{j:(j,i)\in A} x_{ji}\Big) \tag{6.29}$$

$$\text{s.t.}\quad 0 \leqslant x_{ij} \leqslant u_{ij},\quad \forall\,(i,j)\in A. \tag{6.30}$$

式(6.29)可以等价地写成

$$LR(\boldsymbol{\pi}) = \min c(\boldsymbol{x},\boldsymbol{\pi}) = \sum_{(i,j)\in A}(c_{ij} - \pi_i + \pi_j)x_{ij} + \sum_{i\in V} d_i\pi_i$$

$$= \sum_{(i,j)\in A} c_{ij}^{\pi}x_{ij} + \sum_{i\in V} d_i\pi_i. \tag{6.29a}$$

对于给定的势$\boldsymbol{\pi}$，可以很容易得到以上Lagrange松弛问题的最优值$LR(\boldsymbol{\pi})$：如果$c_{ij}^{\pi}>0$，则令$x_{ij}=0$(下界)；如果$c_{ij}^{\pi}<0$，则令$x_{ij}=u_{ij}$(上界)；如果$c_{ij}^{\pi}=0$，则x_{ij}可以是从0到u_{ij}的任意值．也就是说，此时的流$\boldsymbol{x}$和势$\boldsymbol{\pi}$是满足互补松弛条件的．如果$\boldsymbol{x}$又同时是可行流(即满足流量守恒条件(6.1))，则$\boldsymbol{x}$是最小费用流．一般情况下，$\boldsymbol{x}$只是伪流而不是可行流(即满足容量约束(6.30)而不满足流量守恒条件(6.1))，此时得到的$LR(\boldsymbol{\pi})$只是最小费用流问题的一个下界(即$LR(\boldsymbol{\pi})\leqslant z^*$，这里$z^*$表示最小费用流问题的最优值)．松弛算法的思想就是不断改进$\boldsymbol{\pi}$和$\boldsymbol{x}$，在保持互补松弛条件的情况下，使伪流$\boldsymbol{x}$逐步改进为可行流．也就是说，使下界$LR(\boldsymbol{\pi})$不断增加，最后达到z^*．

定义 6.5　对于伪流$\boldsymbol{x}$，为了叙述的方便，我们定义

$$e(i) = d_i + \sum_{j:(j,i)\in A} x_{ji} - \sum_{j:(i,j)\in A} x_{ij}, \tag{6.31}$$

并称 $e(i)$ 为节点 i 的剩余(surplus)或称不平衡数(imbalance). 特别地,当 $e(i)>0$ 时,称 $e(i)$ 为节点 i 的盈余(excess);当 $e(i)<0$ 时,称 $-e(i)$ 为节点 i 的亏空(deficit);当 $e(i)=0$ 时,称节点 i 为平衡点. 可见,当 $\boldsymbol{x}$ 是可行流时,所有节点都是平衡点.

利用记号 $e(i)$,式(6.29)可以等价地写成

$$LR(\boldsymbol{\pi}) = \min c(\boldsymbol{x},\boldsymbol{\pi}) = \sum_{(i,j)\in A} c_{ij}x_{ij} + \sum_{i\in V}\pi_i e(i). \tag{6.29b}$$

与前面的大多数算法一样,松弛算法仍然在残量网络 $N(\boldsymbol{x})$ 上操作. 松弛算法以 $\boldsymbol{\pi}=\boldsymbol{0}$ 和 $\boldsymbol{x}=\boldsymbol{0}$ 作为初始值开始执行,并保持 $\sum_{i\in V} e(i)=0$,即总的盈余与亏空相等. 算法主要包括以下两种基本的改进操作过程:①第1类操作:在势 $\boldsymbol{\pi}$ 不改变的情况下,改进伪流 $\boldsymbol{x}$,使其不可行的程度降低. 具体来讲,就是在 $N(\boldsymbol{x})$ 中找到从一个赢余节点到一个亏空节点的一条增广路,通过增广使得至少赢余节点的盈余严格减少(自然,亏空节点的亏空也严格减少). ②第2类操作:同时修改 $\boldsymbol{\pi}$ 和 $\boldsymbol{x}$,使得 $LR(\boldsymbol{\pi})$ 严格增加.

为了叙述的方便,记 $N(\boldsymbol{x})$ 中 (i,j) 弧上的残留容量为 r_{ij}. 在算法的操作过程中,首先选取一个盈余节点 s,并以 s 为根构造 $N(\boldsymbol{x})$ 中的一棵子树,使得树中的弧 (i,j) 都满足 $c_{ij}^{\pi}=0$,并且树中的节点都是盈余点或平衡点. 记算法执行中某一阶段构造的这棵子树的节点集合为 S,其余的节点集合为 $\bar{S}$. 在 $N(\boldsymbol{x})$ 中,集合 S 决定了一个割 $[S,\bar{S}]$,我们记割中的前向弧为 $(S,\bar{S})$,反向弧为 $(\bar{S},S)$. 由于我们希望将 S 中节点上的赢余减少,因此记

$$e(S) = \sum_{i\in S} e(i), \tag{6.32}$$

$$r(\boldsymbol{\pi},S) = \sum_{(i,j)\in(S,\bar{S})\text{且}c_{ij}^{\pi}=0} r_{ij}. \tag{6.33}$$

如果 $e(S)>r(\boldsymbol{\pi},S)$,则算法可以执行前面所说的第2类操作,即同时修改 $\boldsymbol{x}$ 和 $\boldsymbol{\pi}$,使得 $LR(\boldsymbol{\pi})$ 严格增加. 具体如下:

(1) 算法首先沿 $(S,\bar{S})$ 中满足 $c_{ij}^{\pi}=0$ 的弧增广流量使之饱和,即流量达到残留容量的上界,从而相应的弧不再属于残量网络(其反向弧则可能进入残量网络). 从(6.29a)可以知道,这样对 $\boldsymbol{x}$ 的修改不会改变 $c(\boldsymbol{x},\boldsymbol{\pi})$ 的值,但使得 S 中的节点上的总盈余减少. 由于 $e(S)>r(\boldsymbol{\pi},S)$,因此增广后 S 中的节点上的总盈余 $e(S)-r(\boldsymbol{\pi},S)$ 仍然为正数.

(2) 由于算法保持互补松弛条件,因此对于 $N(\boldsymbol{x})$ 中的任意弧一定有 $c_{ij}^{\pi}\geqslant 0$. 经过增广后,前向弧集 $(S,\bar{S})$ 中的弧一定满足 $c_{ij}^{\pi}>0$,记 c_{ij}^{π} 中的最小值为 $\alpha>0$. 此时,我们将 S 中的每个节点上的势增加 α. 这样的操作只可能使得 $(S,\bar{S})$ 中的弧上的 c_{ij}^{π} 减少 α,使得 $(\bar{S},S)$ 中的弧上的 c_{ij}^{π} 增加 α,其他弧上的 c_{ij}^{π} 不变. 因此,这一操作完成之后,对于 $N(\boldsymbol{x})$ 中的任意弧仍然满足 $c_{ij}^{\pi}\geqslant 0$,即保持互补松弛条件. 从(6.29b)可以知道,这一操作使得 $LR(\boldsymbol{\pi})$ 的值增加 $(e(S)-r(\boldsymbol{\pi},S))\alpha$.

如果 $e(S)\leqslant r(\boldsymbol{\pi},S)$,则算法可以执行前面所说的第1类操作,即修改 $\boldsymbol{x}$,使得 $e(S)$ 严格减少. 具体如下:

由于 $r(\boldsymbol{\pi},S)\geqslant e(S)>0$,因此$(S,\bar{S})$中至少有一条弧$(i,j)$满足 $c_{ij}^{\pi}=0$. 如果 $e(j)\geqslant 0$,则由 S 的构造方法可知节点 j 可以加入到 S 中,并重新开始下一轮迭代. 如果 $e(j)<0$,则算法沿子树中从 s 到 j 的一条增广路增广流量. 从(6.29a)可以知道,这样对 $\boldsymbol{x}$ 的修改不会改变 $c(\boldsymbol{x},\boldsymbol{\pi})$的值,但使得 S 中的节点上的总盈余减少.

松弛算法的具体过程可以详细描述如下:

松弛算法(relaxation algorithm,计算最小费用流 $\boldsymbol{x}$)

STEP0 给出初始势和初始流: $\boldsymbol{\pi}=\mathbf{0},\boldsymbol{x}=\mathbf{0}$.

STEP1 如果网络中不含有任何赢余节点和亏空节点,则已经得到最优解,计算结束;否则在残量网络 $N(\boldsymbol{x})$中,选择一个赢余节点 s,令 $S=\{s\}$,继续下一步.

STEP2 如果 $e(S)>r(\boldsymbol{\pi},S)$,则转 STEP3;否则在$(S,\bar{S})$中选取一条满足 $c_{ij}^{\pi}=0$ 的弧(i,j). 若 $e(j)<0$,则转 STEP4;否则令 $\text{pred}(j)=i,S=S\cup\{j\}$,继续重复 STEP2.

STEP3 首先沿$(S,\bar{S})$中所有满足 $c_{ij}^{\pi}=0$ 的弧(i,j)增广流量使之饱和,然后将 S 中的每个节点上的势增加 $\alpha=\min\{c_{ij}^{\pi}\mid(i,j)\in(S,\bar{S}),r_{ij}>0\}$. 转 STEP1.

STEP4 根据 pred 中记录的节点,确定子树中从 s 到 j 的一条增广路 P. 沿 P 增广流量 $\delta=\min\{e(s),-e(j),\min\{r_{ij}\mid(i,j)\in P\}\}$. 转 STEP1.

上面的算法中 STEP3 执行的是第 2 类操作,STEP4 执行的是第 1 类操作. 从前面的构造过程可以说明算法如果能够终止,则一定求到了最优解. 下面我们说明当给定的数据都是整数时,算法一定在有限步内终止,因此算法是正确的.

例 6.6 用松弛算法计算如图 6.9(a)所示网络中的最小费用流(图中弧上的前一个数字表示弧的容量,后一个数字表示弧上的单位费用;节点上的数字表示供需量).

从初始势和初始流($\boldsymbol{\pi}=\mathbf{0},\boldsymbol{x}=\mathbf{0}$)开始计算. 第 1 次迭代过程为:

STEP1 由于网络中含有赢余节点和亏空节点,所以在残量网络 $N(\boldsymbol{x})=N$ 中选择一个赢余节点 $s=1$,令 $S=\{s\}=\{1\}$,继续下一步.

STEP2 计算知 $e(S)=3,r(\boldsymbol{\pi},S)=0$. 所以 $e(S)>r(\boldsymbol{\pi},S)$,转 STEP3.

STEP3 $(S,\bar{S})$中没有满足 $c_{ij}^{\pi}=0$ 的弧(i,j),因此不必执行增广流量的操作. 计算可知 $\alpha=\min\{c_{12}^{\pi},c_{13}^{\pi}\}=1$,所以令 $\pi_1=\pi_1+\alpha=1$. 转 STEP1.

第 2 次迭代过程为:

STEP1 由于网络中含有赢余节点和亏空节点,所以在残量网络 $N(\boldsymbol{x})=N$ 中选择一个赢余节点 $s=1$,令 $S=\{s\}=\{1\}$,继续下一步.

STEP2 计算知 $e(S)=3,r(\boldsymbol{\pi},S)=r_{13}=2$ (因为此时 $c_{13}^{\pi}=0$). 所以 $e(S)>r(\boldsymbol{\pi},S)$,转 STEP3.

STEP3 $(S,\bar{S})$中满足 $c_{ij}^{\pi}=0$ 的弧只有(1,3),因此执行增广流量的操作得 $x_{13}=2$. 计

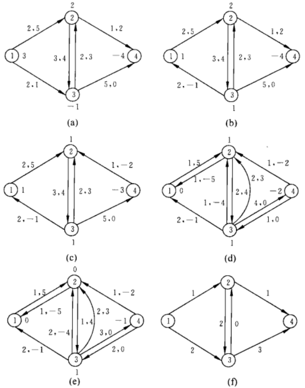


图 6.9 松弛算法计算最小费用流的例子(例 6.6)

算可知 $\alpha=\min\{c_{12}^{\pi}\}=4$,所以令 $\pi_1=\pi_1+\alpha=5$. 转 STEP1.

第 3 次迭代过程为:

STEP1 由于网络中含有赢余节点和亏空节点,所以构造残量网络 $N(\boldsymbol{x})$ 如图 6.9(b).在残量网络 $N(\boldsymbol{x})$ 中选择一个赢余节点 $s=1$,令 $S=\{s\}=\{1\}$,继续下一步.

STEP2 计算知 $e(S)=1, r(\boldsymbol{\pi}, S)=r_{12}=2$ (因为此时 $c_{12}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$ 且 $e(2)=2>0$,所以令 $\text{pred}(2)=1, S=S\cup\{2\}=\{1,2\}$,继续重复 STEP2.

STEP2 计算知 $e(S)=3, r(\boldsymbol{\pi}, S)=0$. 由于 $e(S)>r(\boldsymbol{\pi}, S)$,转 STEP3.

STEP3 计算可知 $\alpha=\min\{c_{24}^{\pi}, c_{23}^{\pi}\}=2$,所以令 $\pi_1=\pi_1+\alpha=7, \pi_2=\pi_2+\alpha=2$. 转 STEP1.

第 4 次迭代过程为:

STEP1 在残量网络 $N(\boldsymbol{x})$ 中选择一个赢余节点 $s=1$,令 $S=\{s\}=\{1\}$,继续下一步.

STEP2 计算知 $e(S)=1, r(\boldsymbol{\pi}, S)=r_{13}=2$ (因为此时 $c_{12}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$ 且

$e(2)=2>0$,所以令 $\text{pred}(2)=1, S=S\cup\{2\}=\{1,2\}$,继续重复 STEP2.

STEP2 计算知 $e(S)=3, r(\boldsymbol{\pi}, S)=0$. 由于 $e(S)>r(\boldsymbol{\pi}, S)$,转 STEP3.

STEP3 $(S,\bar{S})$中满足 $c_{ij}^{\pi}=0$ 的弧只有(2,4),因此沿 2-4 执行增广流量的操作得 $x_{24}=1$. 计算可知 $\alpha=\min\{c_{23}^{\pi}\}=2$,所以令 $\pi_1=\pi_1+\alpha=9, \pi_2=\pi_2+\alpha=4$. 转 STEP1.

第5次迭代过程为:

STEP1 构造残量网络 $N(\boldsymbol{x})$如图 6.9(c).

在残量网络 $N(\boldsymbol{x})$中选择一个赢余节点 $s=1$,令 $S=\{s\}=\{1\}$,继续下一步.

STEP2 计算知 $e(S)=1, r(\boldsymbol{\pi}, S)=r_{12}=2$ (因为此时 $c_{12}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(2)=2>0$,所以令 $\text{pred}(2)=1, S=S\cup\{2\}=\{1,2\}$,继续重复 STEP2.

STEP2 计算知 $e(S)=2, r(\boldsymbol{\pi}, S)=r_{23}=3$ (因为此时 $c_{23}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(3)=1>0$,所以令 $\text{pred}(3)=2, S=S\cup\{3\}=\{1,2,3\}$,继续重复 STEP2.

STEP2 计算知 $e(S)=3, r(\boldsymbol{\pi}, S)=r_{34}=5$ (因为此时 $c_{34}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(4)=-3<0$,所以转 STEP4.

STEP4 根据 pred 中记录的节点,确定子树中从 $s=1$ 到节点 $j=4$ 的一条增广路 $P=$ 1-2-3-4. 沿 P 增广流量 $\delta=\min\{e(s), -e(j), \min\{r_{ij}\mid(i,j)\in P\}\}=e(s)=1$,得 $x_{12}=x_{23}=x_{34}=1$. 转 STEP1.

第6次迭代过程为:

STEP1 构造残量网络 $N(\boldsymbol{x})$如图 6.9(d).

在残量网络 $N(\boldsymbol{x})$中选择一个赢余节点 $s=2$,令 $S=\{s\}=\{2\}$,继续下一步.

STEP2 计算知 $e(S)=1, r(\boldsymbol{\pi}, S)=\min\{r_{21}, r_{23}\}=1$ (因为此时 $c_{21}^{\pi}=c_{23}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(1)=0$,所以令 $\text{pred}(1)=2, S=S\cup\{1\}=\{1,2\}$,继续重复 STEP2.

STEP2 计算知 $e(S)=1, r(\boldsymbol{\pi}, S)=r_{23}=2$ (因为此时 $c_{23}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(3)=1>0$,所以令 $\text{pred}(3)=2, S=S\cup\{3\}=\{1,2,3\}$,继续重复 STEP2.

STEP2 计算知 $e(S)=2, r(\boldsymbol{\pi}, S)=r_{34}=3$ (因为此时 $c_{34}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(4)=-2<0$,所以转 STEP4.

STEP4 根据 pred 中记录的节点,确定子树中从 $s=1$ 到节点 $j=4$ 的一条增广路 $P=$ 2-3-4. 沿 P 增广流量 $\delta=\min\{e(s), -e(j), \min\{r_{ij}\mid(i,j)\in P\}\}=e(s)=1$,得 $x_{23}=x_{34}=2$. 转 STEP1.

第7次迭代过程为:

STEP1 构造残量网络 $N(\boldsymbol{x})$如图 6.9(e).

在残量网络 $N(\boldsymbol{x})$中选择一个赢余节点 $s=3$,令 $S=\{s\}=\{3\}$,继续下一步.

STEP2 计算知 $e(S)=1, r(\boldsymbol{\pi}, S)=\min\{r_{32}, r_{34}\}=2$ (因为此时 $c_{32}^{\pi}=c_{34}^{\pi}=0$). 由于 $e(S)<r(\boldsymbol{\pi}, S)$且 $e(4)=-1<0$,所以转 STEP4.

STEP4 根据 pred 中记录的节点,确定子树中从 $s=3$ 到节点 $j=4$ 的一条增广路 $P=$ 3-4. 沿 P 增广流量 $\delta=\min\{e(s), -e(j), \min\{r_{ij}\mid(i,j)\in P\}\}=e(s)=1$,得 $x_{34}=3$. 转

STEP1.

此时网络中不含有任何赢余节点和亏空节点，已经得到最优解，即最小费用流如图 6.9(f)所示. □

算法复杂度分析：当给定的数据都是整数时，算法每执行 1 次第 2 类操作，则 $LR(\boldsymbol{\pi})$ 至少增加 1 个单位. 算法开始时 $LR(\boldsymbol{\pi})=0$，且算法执行过程中 $LR(\boldsymbol{\pi})$ 不可能超过最小费用的上界 mCU. 因此 STEP3 最多执行 mCU 次. 算法每执行 1 次第 1 类操作，则总赢余至少减少 1 个单位，而总赢余不可能超过其上界 $(m+n)U$，因此在两次连续的第 2 类操作之间，执行第 1 类操作(STEP4)的次数也是有限的，不超过 $(m+n)U=O(mU)$ 次. 进一步，由于执行 1 次第 1 类操作(包括构造增广路的 STEP2 在内)的复杂度为 $O(mn)$，因此算法的总时间复杂度为 $O(mCU\,mU\,mn)=O(nm^3CU^2)$.

从理论分析角度看，这一复杂度结果显然远远高于我们前面介绍的各种算法的复杂度. 但是，大量的计算测试表明，松弛算法的实际计算效率非常高，是目前实际计算效率最高的两个算法之一(另一个算法是网络单纯形方法，我们将在 6.6 节进行介绍). 仔细分析可以看出，松弛算法每次也是沿最小费用路(即满足 $c_{ij}^{\pi}=0$ 的弧 (i,j) 所组成的增广路)增广流量，因此与最小费用路算法有许多共同之处. 但是，松弛算法通过利用额外的中间信息计算 α 来修改对偶变量 $\boldsymbol{\pi}$，也许正是这一中间信息的利用使得松弛算法在实际计算中获得了满意的结果.

松弛算法是由 Bertsekas 等于 20 世纪 80 年代首先提出的，有兴趣的读者可以进一步参看 Bertsekas 的专著[4].

6.6 网络单纯形算法

既然最小费用流问题是一个线性规划问题，它自然应该可以直接采用线性规划算法，如单纯形算法求解. 但是，如果不考虑这一规划问题的网络特性，而是直接调用求解一般线性规划问题的单纯形算法来求解网络优化问题，计算实验表明算法的效率很差(即速度很慢). 有趣的是，如果仔细分析网络优化问题的网络特性，则可以设计出特殊的单纯形算法，一般称为网络单纯形算法(network simplex algorithm). 计算实验表明，网络单纯形算法在实用中的计算效率很高(即速度很快). 本节介绍这一算法.

线性规划问题的单纯形算法的一般思路是：首先求得问题的一个基本可行解，如果它不是最优解，则选择一个进基变量(列)和一个出基变量(列)，通过旋转(pivot)变换改进到另一个基本可行解，如此反复迭代，直到检验数(或称相对费用，reduced cost)都大于等于 0 为止，获得最优解. 可以看出，单纯形算法始终保持解的原始可行性，但不一定对偶可行；一旦达到最优解，则同时获得可行对偶解，且互补松弛条件成立. 算法的计算过程通常利用单纯形表进行.

基本可行解是由基矩阵所决定的，所以基矩阵在单纯形算法中是非常关键的．为了利用单纯形算法求解最小费用流问题，我们首先必须清楚线性规划问题的基的结构．对于网络优化问题，基的结构是非常特殊的，因此可以避免显式地列出单纯形表．

不失一般性，我们在以下讨论中总是假设流网络在不考虑弧的方向时是连通的（否则该问题可以自然地分解为几个小问题来考虑）．

6.6.1 算法的一般思路

我们首先考虑容量无限的最小费用流问题，即假设所有弧上的容量下界为0，上界为无穷大．对于这一特殊问题，利用关联矩阵 $\boldsymbol{B}$，式(6.1)～式(6.3)可以表示为：

$$\min \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \tag{6.34}$$

$$\text{s.t. } \boldsymbol{Bx}=\boldsymbol{d} \tag{6.35}$$

$$\boldsymbol{x}\geqslant\boldsymbol{0} \tag{6.36}$$

很清楚，约束条件(6.35)包含 n 个等式约束，其中只有 $n-1$ 个约束是独立的，因此可以任意去掉其中的某一个约束．我们在后面的叙述中不再对此加以特别说明．

引理 6.8 关联矩阵 $\boldsymbol{B}$ 的列构成一组基的充要条件是它所对应的弧为支撑树．

证明 因为本节我们已经假设流网络在不考虑方向时是连通的，所以 $\mathrm{r}(\boldsymbol{B})=n-1$．记 $\boldsymbol{B}$ 的 $n-1$ 列构成的子矩阵为 $\boldsymbol{B}_1$．

必要性：若 $\boldsymbol{B}_1$ 为一组基，则 $\mathrm{r}(\boldsymbol{B}_1)=n-1$．$\boldsymbol{B}_1$ 所对应的 $n-1$ 条弧如果不连通（不考虑方向），则至少含有一个圈，因此 $\mathrm{r}(\boldsymbol{B}_1)<n-1$，矛盾．因此 $\boldsymbol{B}_1$ 所对应的 $n-1$ 条弧一定是连通的（不考虑方向），即这些弧构成一棵支撑树．

充分性：$\boldsymbol{B}_1$ 所对应的 $n-1$ 条弧构成一棵支撑树，则 $\mathrm{r}(\boldsymbol{B}_1)=n-1$．因此 $\boldsymbol{B}_1$ 是一组基．

□

那么，当给定了基 $\boldsymbol{B}_1$，如何确定变量的值（弧上的流）呢？按照一般单纯形算法的表示方法，基变量的值为 $\boldsymbol{B}_1^{-1}\boldsymbol{d}$．这里我们把 $\boldsymbol{B}_1$ 所对应的弧集合（支撑树）用 T 表示，则所有非树弧（非基弧）对应于非基变量，其上的流量为0；只有 T 中的弧（树弧，或称基弧）对应于基变量，其上的流量可以为正数（在退化的情况下也可能为0）．于是，确定基变量的值的问题在网络上等价于：当给定了支撑树 T 之后，确定树弧上的流量并使之满足流量守恒条件(6.35)．

对于 T 中的一条弧 (i,j)，它总是将 T 分解为两棵子树 T_i 和 T_j，并且 i,j 分别属于 T_i 和 T_j．由于 $\sum_{k\in T_i}d_k$ 是子树 T_i 中所有节点上的供需量之和，而弧 (i,j) 是子树 T_i 和 T_j 之间的唯一连接弧，所以为了使得节点 i 和 j 流量守恒，弧 (i,j) 上的流量必须是 $\sum_{k\in T_i}d_k\left(\text{由于}\sum_{k\in V}d_k=0\text{，所以}\sum_{k\in T_i}d_k=-\sum_{k\in T_j}d_k\right)$．根据这样一个性质，我们可以得到一个计算树弧上的流量的快速算法：对于树弧 (i,j)，假设它的其中一个节点是叶子节点（不妨设

为节点 j)，则弧(i,j)上的流量必须是 $-d_j$. 当弧(i,j)上的流量确定后，我们可以将(i,j)从 T 中删去(自然，节点 i 上的供需量应当相应地进行修改)，然后重复这一过程，直到所有树弧上的流量都得到确定为止.

但是，上述过程所确定的流 $\boldsymbol{x}$ 仍然不一定是可行的，即某些树弧上的流量可能为负数. 我们把使得相应的流 $\boldsymbol{x}$ 为可行流的基 $\boldsymbol{B}_1$ 称为可行基，支撑树 T 称为可行树. 与单纯形算法只对基本可行解操作对应，网络单纯形算法只对可行树进行操作. 为了判断什么时候可行树所对应的流是最优流，算法仍然利用检验数(相对费用)进行判断. 按照一般单纯形算法的表示方法，检验数的值为 $\boldsymbol{C}-\boldsymbol{\pi B}$，这里 $\boldsymbol{\pi}$ 为对偶变量. 对于最小费用流问题，(i,j)弧对应的检验数的值为 $c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j$($\boldsymbol{\pi}$ 为对偶变量，即节点上的势). 下面讨论如何快速地获得对偶变量 $\boldsymbol{\pi}$ 的值.

在容量无限这一特殊的问题中，互补松弛条件(6.19)～(6.21)为

$$\text{当 } x_{ij}=0 \text{ 时，}\quad c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j\geqslant 0; \tag{6.37}$$

$$\text{当 } x_{ij}>0 \text{ 时，}\quad c_{ij}^{\pi}=c_{ij}-\pi_i+\pi_j=0. \tag{6.38}$$

假设我们已经给定了一个基本可行解 $\boldsymbol{x}$，基矩阵所对应的可行树为 T. 由于只有树弧上的流量可以为正数，所以只有树弧才可能满足(6.38). 支撑树上的弧共有 $n-1$ 条，而对偶变量(节点上的势)共有 n 个. 在相差一个常数的意义下，由 T 中的弧满足 $c_{ij}-\pi_i+\pi_j=0$ 可以唯一地确定对偶变量(节点上的势 π_i). 具体来说，我们可以任意选定一个节点(这一节点通常称为"根"(root))，令它的势为 0；然后利用(6.38)计算与它相邻的其他节点上的势，如此重复就可以方便地获得所有节点上的势. 如果如此确定的对偶变量 π_i 同时使得(6.37)也成立，则 $\boldsymbol{x}$ 就是原问题的最优解. 实际上，约束(6.37)表示的就是检验数非负这一最优条件，或者说正是 $\boldsymbol{\pi}$ 为对偶可行解的要求.

反过来说，如果 $\boldsymbol{x}$ 不是最优解，则一定存在一条非树弧(p,q)使得(6.37)不成立，即 $c_{pq}^{\pi}=c_{pq}-\pi_p+\pi_q<0$. 此时，弧$(p,q)$可以进入基. 为了找出原来基中的一条弧出基，我们可以看出 $T\cup\{(p,q)\}$ 一定含有一个唯一的圈 W，我们把弧(p,q)的方向定义为 W 的方向. W 的费用为

$$\begin{aligned}
C(W) &= \sum_{(i,j)\in W^+} c_{ij} - \sum_{(i,j)\in W^-} c_{ij} \\
&= \sum_{(i,j)\in W^+} (c_{ij}-\pi_i+\pi_j) - \sum_{(i,j)\in W^-} (c_{ij}-\pi_i+\pi_j) \\
&= c_{pq}^{\pi},
\end{aligned}$$

即 W 为一个负费用圈，所以沿 W 增广流量将会使得总费用下降. 为了在 W 中找出一条弧出基，我们应当令增广的流量等于 W^- 所有弧上当前流量中的最小值，而取到该最小值的弧出基(如果 $W^-=\varnothing$，则原问题是无界的，即最小费用可以趋于负无穷大).

基于以上讨论，我们对网络单纯形算法的一般思路概括如下：

网络单纯形算法求解容量无限的最小费用流问题

STEP0 获得一个初始的可行树 T 及对应的基本可行解 $\boldsymbol{x}$(我们将在后面介绍具体方法).

STEP1 计算对偶变量 $\boldsymbol{\pi}$.

STEP2 判断是否为最优解,若是,则停止;否则选定一个进基变量(即选进基弧 (p,q)).

STEP3 选定一个出基变量(即选出基弧),如果找不到这样的弧,则原问题是无界的,停止;否则进行下一步.

STEP4 设 W 为 $T\cup\{(p,q)\}$ 所含的圈.沿 W 的正向增广流量,即修改 $\boldsymbol{x}$ 及对应的可行树 T,回到 STEP1.

例 6.7 用网络单纯形算法计算如图 6.10(a)所示网络中的最小费用流(图中弧上的数字表示弧上的单位费用;节点上的数字表示供需量;假设所有弧的下容量为 0,上容量为无穷).

首先假设已经获得一个初始的可行树 $T=\{(1,2),(1,3),(2,4)\}$.对应的基本可行解 $\boldsymbol{x}$ 和对应的对偶变量 $\boldsymbol{\pi}$ 如图 6.10(b)所示(图中弧上的数字表示树弧上的流量 $\boldsymbol{x}$;令节点 4 的对偶变量为 0,节点上的数字表示对偶变量 $\boldsymbol{\pi}$).当前费用为 $6+20-5=21$.对于非树弧计算:

$$c_{23}^{\pi}=-1-4+11=6>0,$$
$$c_{32}^{\pi}=6-11+4=-1<0,$$
$$c_{34}^{\pi}=3-11+0=-8<0.$$

选取(3,2)为进基弧,负费用圈 $W=\{(3,2),(1,2),(1,3)\}$.选取(1,2)为出基弧,沿 W 增广 3 个单位流量,修改对偶变量 $\boldsymbol{\pi}$,得到如图 6.10(c)所示网络.当前费用为 $-20+18+20=18$.对于非树弧计算:

$$c_{12}^{\pi}=2-5+4=1>0,$$
$$c_{23}^{\pi}=-1-4+10=5>0,$$
$$c_{34}^{\pi}=3-11+0=-8<0.$$

选取(3,4)为进基弧,负费用圈 $W=\{(3,4),(2,4),(3,2)\}$.选取(1,2)为出基弧,沿 W 增广 3 个单位流量,修改对偶变量 $\boldsymbol{\pi}$,得到如图 6.10(d)所示网络.当前费用为 $-20+9+8=-3$.对于非树弧计算:

$$c_{12}^{\pi}=2-(-2)+4=8>0,$$
$$c_{23}^{\pi}=-1-4+3=-2<0,$$
$$c_{32}^{\pi}=6-3+4=7>0.$$

选取(2,3)为进基弧,负费用圈 $W=\{(2,3),(3,4),(2,4)\}$.选取(2,4)为出基弧,沿 W 增广 2 个单位流量,修改对偶变量 $\boldsymbol{\pi}$,得到如图 6.10(e)所示网络.当前费用为 $-20+15+$

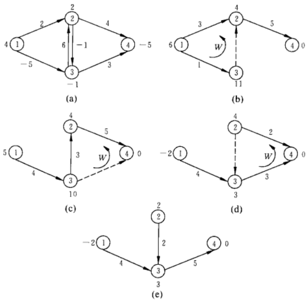


图 6.10　松弛算法计算最小费用流的例子(例 6.7)

(−2)=−7. 对于非树弧计算：

$$c_{12}^{\pi}=2-(-2)+2=6>0,$$
$$c_{32}^{\pi}=6-3+2=5>0,$$
$$c_{24}^{\pi}=4-2+0=2>0.$$

此时得到最优解. □

6.6.2　处理退化的方法

我们在线性规划中学过，一般的单纯形算法是可能发生退化甚至循环的. 所谓退化，是指某一旋转变换增广的流量为 0(即出弧上的流量在增广前已经为 0)，因此并不真正获得费用的改进. 所谓循环，是指存在一个旋转变换的周期，此周期内的旋转变换都是退化的，因此算法无法终止. 在网络单纯形算法中，如果不进行进一步的特殊处理，则也可能会出现退化甚至循环. 实际计算测试研究表明，网络单纯形算法中往往 90% 以上的旋转变换都是退化的. 所以，谨慎地处理退化对保证算法的效率是非常关键的.

我们说过，网络单纯形算法中实际上是对可行树进行操作. 如果算法能够保证每次所操作的可行树"都不相同"，则算法一定不会出现循环，从而一定在有限步内终止. 这里，所谓的"树不相同"，是指节点已经标号的树不相同(也就是说，所谓不相同的树，并不排除他们是同构的).

这里我们介绍处理退化的强可行树(strongly feasible spanning tree)方法.

定义 6.6　假定计算节点上的势时所选定的“根节点”是固定的.对于可行树 T 中的一条树弧(i,j),如果 T 中从根到 j 的路通过节点 i,则(i,j)称为远离根节点的弧(downward pointing arc).如果 T 中的所有流量为 0 的弧都是远离根节点的弧,则称可行树 T 为强可行树.

例 6.8　如图 6.11 所示网络中,假设弧上的数字表示当前可行流.若节点 1 为根节点,则 $T_1=\{(1,3),(3,2),(3,4)\}$为强可行树,$T_2=\{(1,3),(2,3),(3,4)\}$不是强可行树,因为 T_2 中$(2,3)$不是远离根节点的弧.　□

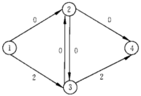


图 6.11　强可行树的例子(例 6.8)

引理 6.9　如果网络单纯形算法中生成的所有可行树都是强可行树,则这些树互不相同.

证明　如果网络单纯形算法中的旋转变换不是退化的,则相应的可行树对应的可行流费用互不相同,因此这些树也一定互不相同.所以,我们只需要考虑旋转变换退化的情况.如果 T 为生成树,r 为根节点,记 $\pi(T)=\sum_{i\in V}(\pi_r-\pi_i)$.

考虑算法过程中连续生成的两棵强可行树 $T,\overline{T}=T\cup\{(p,q)\}\setminus\{(k,l)\}$,即$(p,q)$为进基弧,$(k,l)$为出基弧,且从 T 到 $\overline{T}$ 的旋转是退化的.旋转变换前后,(p,q)弧上的流量都是 0,由于 $\overline{T}$ 是强可行树,所以(p,q)弧一定是远离根节点 r 的弧.(p,q)弧将 $\overline{T}$ 分解为两棵子树 T_p 和 T_q,并且 p,q 分别属于 T_p 和 T_q,则 $r\in T_p$.记 $\overline{T}$ 对应的势为 $\bar{\pi}$,根据势的确定方法,可以得到(留作练习):

$$\text{当 } i\in T_p \text{ 时},\quad \bar{\pi}_i=\pi_i;\tag{6.39}$$

$$\text{当 } i\in T_q \text{ 时},\quad \bar{\pi}_i=\pi_i-c_{pq}^{\pi};\tag{6.40}$$

因为 $c_{pq}^{\pi}<0$,所以

$$\bar{\pi}(\overline{T})=\pi(T)+|T_q|c_{pq}^{\pi}<\pi(T).$$

因此,T 和 $\overline{T}$ 是两棵不同的强可行树.　□

上述引理说明,如果网络单纯形算法中生成的所有可行树都是强可行树,则可以避免发生循环.那么,如何保证只对强可行树进行处理呢?首先,我们当然要求初始的基本可行解是对应于一棵强可行树,这一点我们将在后面详细介绍;其次,我们要求旋转变换只生成强可行树.这就要求算法在选择出基弧时不能完全没有任何限制,而是要有一定的优先顺序.设旋转过程之前,强可行树为 T,(p,q)为进基弧,$T\cup\{(p,q)\}$包含的圈为 W.假设 $W^-\neq\varnothing$,令 $\delta=\min\{x_{ij}\mid(i,j)\in W^-\}$,$\overline{W}=\{(i,j)\in W^-\mid x_{ij}=\delta\}$,则 $\overline{W}$ 为所有可能的出基弧的集合.进一步,我们记节点 $\bar{p}$ 是 T 中从根节点 r 到节点 p 的路与圈 W 的第 1 个交点,并选取出基弧(k,l)为从 $\bar{p}$ 出发沿圈 W 的正向前进时第 1 次所遇到的 $\overline{W}$ 中的弧(前面说过,我们把弧(p,q)的方向定义为 W 的方向).

很容易看出 $\overline{T}=T\cup\{(p,q)\}\setminus\{(k,l)\}$ 仍然是强可行树. 如果旋转是非退化的,则我们只需要检验 $\overline{W}\setminus\{(k,l)\}$ 中的弧在 $\overline{T}$ 中是否为远离根节点的弧;如果旋转是退化的,则我们只需要检验 $\overline{W}\cup\{(p,q)\}\setminus\{(k,l)\}$ 中的弧在 $\overline{T}$ 中是否为远离根节点的弧. 由于出基弧 (k,l) 为从 $\hat{p}$ 出发沿圈 W 的正向前进时第 1 次所遇到的 $\overline{W}$ 中的弧,所以上述问题的答案都是肯定的,即 $\overline{T}$ 是强可行树.

6.6.3　初始的基本可行解

前面的讨论中我们没有说明如何获得第 1 个(初始)基本可行解,并且为了处理退化,我们希望它对应的可行树是强可行树. 一般可以采用大-M 方法(big-M method)构造初始的强可行树.

从线性规划的学习中知道,在大-M 方法中,需要加入一系列人工变量(人工弧). 这里我们首先在原问题的网络中加入一个人工节点 0,并假设其供需量为 $d_0=0$. 然后对原网络中的所有节点 i,按如下步骤加入人工弧: 如果 $d_i>0$,则加入人工弧$(i,0)$;否则加入人工弧$(0,i)$. 记所有人工弧的集合为 $T^{(0)}$,所有人工弧上的费用假定为一个充分大的正数 M,并称原网络加入人工弧后的新网络为人工网络.

显然,$T^{(0)}$ 是人工网络的一棵强可行树,因此可以对人工网络上的最小费用流问题直接应用网络单纯形算法求解. 由于 M 是一个充分大的正数,因此当算法终止时,有 3 种情况:

(1) 人工网络上的最小费用流问题有有界的最优解,且最优解中所有人工弧上的流量为 0,则原问题也有有界的最优解,且人工网络上非人工弧上的流量正好就是原问题的最优解.

(2) 人工网络上的最小费用流问题有有界的最优解,且最优解中某些人工弧上的流量不为 0,则原问题是不可行的.

(3) 人工网络上的最小费用流问题没有有界的最优解(即最优值趋向负无穷),则原问题也没有有界的最优解(即最优值趋向负无穷).

然而,如果 M 的取值不足够大,即使原问题有有界的最优解,人工网络上的最小费用流问题可能也会没有有界的最优解(即最优值趋向负无穷). 那么,实际计算中 M 取多大才是"充分大"? 理论上可以证明,当 $M>(n-1)C/2$ 时,上述结论(1)～(3)就是成立的(这里 $C=\max\{|c_{ij}|:(i,j)\in A\}$). 在实用中,一般采用一种自适应策略: 首先取 M 为某一个中等规模大小的正数进行计算;如果最优解中某些人工弧上的流量不为 0,则增加 M 的规模重新计算.

6.6.4　容量有界的情形

网络单纯形算法可以直接推广到容量有界的最小费用流问题,即假设所有弧上的容量可以有下界和上界的约束. 这里我们直接给出相应的结论,详细的推导留给读者自己完成. 首先,这一一般问题可以表示为:

$$\min \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \tag{6.41}$$

$$\text{s.t.}\ \boldsymbol{Bx}=\boldsymbol{d} \tag{6.42}$$

$$l_{ij} \leqslant x_{ij} \leqslant u_{ij} \tag{6.43}$$

在前面的讨论中，我们用支撑树来表示基，只有树弧上的流量可以为正数，所有非树弧上的流量等于下界 0. 对于容量有上下界约束的情形，仍然只有树弧上的流量可以不等于下界和上界，而所有非树弧上的流量只能等于下界或上界. 但是，与前面的讨论不同的是，这里我们对非树弧还需要进一步明确它的流量究竟等于下界还是上界. 也就是说，基可以用所有弧的一个划分(T,L,U)来表示，其中 T 是一棵支撑树，L 是非树弧中流量等于下界的弧的集合，U 是非树弧中流量等于上界的弧的集合. 三元组(T,L,U)可以称为基结构(有时也直接简称为基)，或支撑树结构. 给定一个基结构(T,L,U)，非树弧上的流量已经确定，所以树弧上的流量也可以方便地根据节点上的流量守恒约束计算出来，并且也是唯一的. 如果这些流量同时满足容量的上下界约束，则(T,L,U)是可行支撑树结构(简称可行树结构). 节点上的势也可以与前面的讨论完全类似地进行计算.

为了处理退化和循环，仍然可以类似强可行树定义强可行树结构.

定义 6.7 假定计算节点上的势时所选定的"根节点"是固定的. 在可行树结构(T,L,U)中，如果树弧中所有流量等于下界的弧都是远离根节点的，并且树弧中所有流量等于上界的弧都不是远离根节点的(可以称为面向根节点的)，则(T,L,U)是强可行树结构.

为了从一个初始的强可行树结构开始迭代，仍然可以构造人工网络，采用大-M 方法. 具体方法是：增加人工节点 0，并假设其供需量为 $d_0=0$. 然后对原网络中的所有节点 i，按如下步骤加入人工弧：如果 $d_i>0$，则加入人工弧$(i,0)$；否则加入人工弧$(0,i)$. 记所有人工弧的集合为 $T^{(0)}$，所有人工弧上的费用假定为一个充分大的正数 M，并称原网络假入人工弧后的新网络为人工网络. 此时，问题转化为

$$\min c(\boldsymbol{x}) = \sum_{(i,j)\in A} c_{ij}x_{ij} + \mathrm{M}\Big(\sum_{(i,0)\in T^{(0)}} x_{i0} + \sum_{(0,i)\in T^{(0)}} x_{0i}\Big) \tag{6.44}$$

$$\text{s.t.}\ \sum_{j:(i,j)\in A\cup T^{(0)}} x_{ij} - \sum_{j:(j,i)\in A\cup T^{(0)}} x_{ji} = d_i,\quad \forall\, i \in V \cup \{0\}, \tag{6.45}$$

$$l_{ij} \leqslant x_{ij} \leqslant u_{ij},\quad \forall\,(i,j)\in A, \tag{6.46}$$

$$0 \leqslant x_{i0} \leqslant \bar{d}_i,\quad \forall\, i \text{ 且 } d_i > 0, \tag{6.47}$$

$$0 \leqslant x_{0i} \leqslant \underline{d}_i,\quad \forall\, i \text{ 且 } d_i \leqslant 0, \tag{6.48}$$

其中

$$\bar{d}_i = d_i - \sum_{j:(i,j)\in A} l_{ij} + \sum_{j:(j,i)\in A} l_{ji}, \tag{6.49}$$

$$\underline{d}_i = -d_i + \sum_{j:(i,j)\in A} l_{ij} - \sum_{j:(j,i)\in A} l_{ji}. \tag{6.50}$$

此时，$(T^{(0)},A,\varnothing)$是初始的强可行树结构，且初始基本可行解为：$\forall\,(i,j)\in A, x_{ij}=l_{ij}$；$\forall\,(i,0)\in T^{(0)}$ 即 $d_i>0, x_{i0}=\bar{d}_i$；$\forall\,(0,i)\in T^{(0)}$ 即 $d_i\leqslant 0, x_{0i}=-d_i$.

下面我们直接给出网络单纯形算法的过程：

一般的网络单纯形算法

STEP0　获得一个初始的基本可行解 $\boldsymbol{x}$ 及对应的强可行树结构(T,L,U).

STEP1　计算对偶变量$\boldsymbol{\pi}$.

STEP2　判断是否达到最优解. 根据互补松弛条件，如果$\forall (i,j)\in L, c_{ij}^{\pi}\geqslant 0$；并且$\forall (i,j)\in U, c_{ij}^{\pi}\leqslant 0$，则已经达到最优解. 否则选取一条不满足上述条件的弧(p,q)为进基弧.

STEP3　设 $T\cup\{(p,q)\}$包含的圈为 W. 如果$(p,q)\in L$，我们把弧(p,q)的方向定义为 W 的方向（正向）. 如果$(p,q)\in U$，我们把弧(p,q)的相反方向定义为 W 的正向. 假设 $W^-\neq\varnothing$，令 $\delta=\min(\min\{x_{ij}-l_{ij}\mid(i,j)\in W^-\},\min\{u_{ij}-x_{ij}\mid(i,j)\in W^+\})$，$\overline{W}=\{(i,j)\in W^-\mid x_{ij}-l_{ij}=\delta\}\cup\overline{W}=\{(i,j)\in W^+\mid u_{ij}-x_{ij}=\delta\}$，则 $\overline{W}$ 为所有可能的出基弧的集合. 进一步，我们记节点 $\bar{p}$ 是 T 中从根节点 r 到节点 p 的路与圈 W 的第一个交点，并选取出基弧(k,l)为从 $\bar{p}$ 出发沿圈 W 的正向前进时第一次所遇到的 $\overline{W}$ 中的弧.

STEP4　沿圈 W 的正向增广流量 δ，即修改 $\boldsymbol{x}$ 及对应的强可行树结构(T,L,U)，回到 STEP1.

例 6.9　用网络单纯形算法计算如图 6.12(a)所示网络中的最小费用流（假设所有弧上的下界为 0，图中弧上的前一个数字表示弧的上容量，后一个数字表示弧上的单位费用；节点上的数字表示供需量）.

首先假设已经获得一个初始的可行树结构 $T=\{(1,2),(1,3),(2,4),(2,5),(5,6)\}$，$L=\{(2,3),(5,4)\}$，$U=\{(3,5),(4,6)\}$. 对应的基本可行解 $\boldsymbol{x}$ 和对应的对偶变量 $\boldsymbol{\pi}$ 如图 6.12(b)所示（图中弧上的数字表示树弧上的流量 $\boldsymbol{x}$；令节点 3 的对偶变量为 0，节点上的数字表示对偶变量 $\boldsymbol{\pi}$）.

此时，树弧中只有(1,2)上的流量是等于其上容量的，且(1,2)是面向根节点的，因此(T,L,U)是强可行树结构. 计算可知 $c_{35}^{\pi}=4-0+(-3)=1>0$，因此它不满足最优条件，我们选取(3,5)为进基弧. 此时，沿增广圈可以增广的最大流量为 1，即(2,5)出基. 增广并计算对偶变量得到如图 6.12(c)所示网络.

此时，可行树结构 $T=\{(1,2),(1,3),(2,4),(3,5),(5,6)\}$，$L=\{(2,3),(5,4)\}$，$U=\{(2,5),(4,6)\}$. 由于树弧上的流量都不等于其上下容量，因此$(T,L,U)$仍然是强可行树结构. 计算可知 $c_{46}^{\pi}=3-(-6)+(-8)=1>0$，因此它不满足最优条件，我们选取(4,6)为进基弧. 此时，沿增广圈可以增广的最大流量为 1，即(3,5)出基. 增广并计算对偶变量得到如图 6.12(d)所示网络.

容易验证，此时已经得到最优解.　□

值得指出的是，上面介绍的网络单纯形算法 STEP3 中选取的出基弧可能就是在 STEP2 中选取的进基弧. 这时，经过一次旋转，虽然 T 是不变的，但 L,U 会发生变化，算法

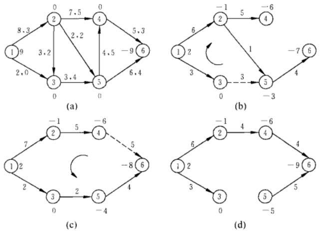


图 6.12 网络单纯形算法计算最小费用流的例子(例 6.9)

过程中只对强可行树结构进行处理,因此可以避免出现循环.为了获得较好的计算效率,在计算机上具体实现网络单纯形算法时,一般还要仔细选定具体的网络表示方式,以及对树的操作采用一些特殊的高效算法等,这些细节我们就不再详细介绍了,有兴趣的读者可以参看一些有关专著(如参考书目[1][4]).

虽然网络单纯形算法在实际计算中效率很高,但是从理论上分析,一般来说网络单纯形算法不是多项式时间算法.利用强可行树结构虽然可以避免出现循环,保证算法在有限步内结束,但连续出现退化旋转的次数仍然可能是指数的.虽然本章我们介绍的算法都不是多项式时间算法,但求解最小费用流问题的多项式时间算法是存在的.实际上,从20世纪70年代以来,尤其是在20世纪80年代和20世纪90年代,已经发表了许多求解最小费用流问题的多项式时间算法.这些算法一般都要用到所谓"变尺度"技术,对我们在本章中介绍的算法进行一些改进,并采用一些特殊的数据结构和实施技巧等.由于介绍其细节和证明其多项式性比较烦琐,而实际计算中这些算法的效果优势并不明显,我们就不再详细介绍了,有兴趣的读者可以参看一些有关专著.目前,求解最小费用流问题的多项式时间算法中,复杂度较低的几个算法的最坏时间界为: $O(nm\log(n^2/m)\log(nC))$, $O(nm(\log\log U)\log(nC))$ 和 $O((m\log n)(m+n\log(nC)))$ 等.

练 习 题

1. 某商场采购某种产品用以满足市场需求,且已知在时段 t 中的市场需求为 $d_t(t=1,2,\cdots,T)$,单件销售价为 p_t. 在某时段 t 之初,商场采购量至多为 u_t,单件采购价为 w_t;在某时段 t 之末,商场库存量至多为 s_t,单件库存费用为 h_t. 商场应如何安排采购计划,可以保证

按时满足需求，且使总利润最大？请将该问题建模为一个最小费用流问题.

2. 假设某个城市分为 L 个区，每个区有若干男孩和若干女孩需要上学. 假设每个区有一所小学，每所小学所能容纳的学生总数已知，并且按照规定，每所小学所能容纳的男孩和女孩比例不能太大或太小. 假设每两个区之间的路程已知（同一区内认为路程近似为 0），如何为需要上学的小孩分配学校，使得所有小孩上学所走的总路程最少？请将该问题建模为一个最小费用流问题.

3. 在线性规划问题中，假设约束矩阵是 0-1 矩阵，且约束矩阵的每一列中，1 是连续出现的. 如何把这一线性规划问题转化为一个最小费用流问题？如果约束矩阵的每一行中，1 是连续出现的，是否可以把这一线性规划问题转化为一个最小费用流问题？

4. 请给出如下的等价变换过程，并说明该变换过程是否可在多项式时间内完成：

(1) 一般网络的最小费用流问题，可以等价地变换为单源单汇网络的最小费用流问题.

(2) 容量下界不是 0 的最小费用流问题，可以等价地变换为容量下界为 0 的最小费用流问题.

(3) 具有容量上界的最小费用流问题，可以等价地变换为没有容量上界（即容量上界是正无穷）的最小费用流问题.

(4) 一般的最小费用流问题，可以等价地变换为最小费用循环流问题.

(5) 一般的最小费用流问题，可以等价地变换为运输问题.

(6)一般的最小费用流问题，可以等价地变换为指派问题.

5. 证明下列说法正确或给出反例说明其不成立.

(1) 如果将有向流网络中所有弧上的费用同时扩大 k 倍（$k>0$），则最小费用流不变.

(2) 如果将有向流网络中所有弧上的费用同时减少 k 个单位（$k>0$），则最小费用流不变.

(3) 如果 (p,q) 是有向流网络的所有弧中费用最小的弧，则一定存在某个最小费用流，使得 (p,q) 上的流量为正数（假设所有弧上的下容量为 0）.

(4) 如果 (p,q) 是有向流网络的所有弧中费用最大的弧，则不存在最小费用流，使得 (p,q) 上的流量为正数（假设所有弧上的下容量为 0）.

(5) 如果最小费用流问题中的所有容量和供需量都是偶数，则一定存在某个最小费用流，使得所有弧上的流量为偶数.

(6) 如果最小费用流问题中的所有容量和供需量都是奇数，则一定存在某个最小费用流，使得所有弧上的流量为奇数.

6. 在最小费用流问题中，假设所有弧上的下容量为 0，某些弧没有上容量限制（即上容量为无穷），某些弧上的单位费用为负数. 记 U 为所有上容量不为无穷的弧上的上容量以及所有源点上的供应量的和.

(1) 证明：该问题有有界的最优解当且仅当没有上容量限制的弧不构成任何负费用有向圈.

(2) 证明：如果问题有有界的最优解，则存在所有弧上流量均不超过 U 的最优解.

(3) 假设将没有上容量限制的所有弧上的上容量改为 U 以后，得到最优值 z；而改为 $U+1$ 以后，得到最优值 z'. 证明：原问题没有有界的最优解当且仅当 $z'<z$.

7. 如果最小费用流问题中的所有容量和供需量都是整数，且已经得到了某个不是整流的最小费用流. 设计一个算法，把该最小费用流转化为一个最小费用整流(要求转化中每一步都保持最优性). 分析你给出的算法的复杂度.

8. 证明：第6.4节中有上下容量约束的最小费用流问题的互补松弛条件为(6.23)～(6.25).

9. 给出反例说明下列结论不成立：如果最小费用流问题中的所有费用都是正数，则当一对相邻的源点和汇点上的供需量都增加 k 个单位($k>0$)时，最小费用流的费用应当增加.

10. 证明下列说法正确：在某最小费用流问题中，如果存在两个最小费用流，它们在 (p,q) 弧上的流量分别为 x 和 y，则对于 x 和 y 的任意凸组合 z，存在某个最小费用流，它在 (p,q) 弧上的流量为 z.

11. 用消圈算法计算如图6.13所示网络中的最小费用流(图中弧上的下容量为0，前一个数字表示弧上的上容量，后一个数字表示弧上的单位费用；节点上的数字表示供需量).

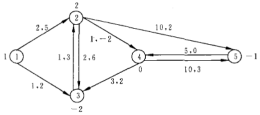


图6.13 第11题图

12. 用最小费用路算法计算练习题11网络中的最小费用流.

13. 在如图6.14所示网络中，图中弧上的下容量为0，前一个数字表示弧上的上容量，后一个数字表示弧上的单位费用，节点1为唯一的源点，节点6为唯一的汇点. 用最小费用路算法计算：

(1) 从节点1流到节点6的流值为2的最小费用流；

(2) 从节点1流到节点6的最小费用最大流；

(3) 从节点1流到节点6的费用不超过29的最大流.

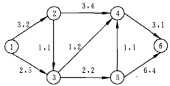


图6.14 第13题图

14. 在最小费用流问题中，如果弧上的费用不是整数(但问题的其他参数仍然是整数)，最小费用路算法是否一定在有限步内结束？如果弧上的容量也不是整数，最小费用路算法是否一定在有限步内结束？如果是，请说明理由；如果不是，说明算法做怎样的修改可以保证一定在有限步内结束.

15. (1) 分别用消圈算法(每次沿最负的负费用圈增广)和最小费用路算法计算如

图 6.15 所示网络中的最小费用流(图中弧上的下容量为 0,前一个数字表示弧上的上容量,后一个数字表示弧上的单位费用;节点上的数字表示供需量).

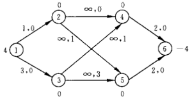


图 6.15　第 15 题图

(2) 证明:如果初始化适当,则消圈算法(每次沿最负的负费用圈增广)和最小费用路算法是等价的,即它们以同样的顺序沿同样的增广路增广.

16. 用原始-对偶算法计算练习题 11 网络中的最小费用流.

17. 针对原始-对偶算法,考虑与练习题 14 相同的问题.

18. 在最小费用流问题中:

(1) 如果某些弧上的下容量为负数,能否使用瑕疵算法求解?

(2) 如果某些弧上的下容量等于上容量,能否使用瑕疵算法求解?

(3) 如果某些节点不是转运点(即问题不是针对循环流),能否使用瑕疵算法求解?

19. 用瑕疵算法计算如图 6.16 所示网络中的最小费用流(图中弧上的前两个数字分别表示弧的下容量和上容量,第 3 个数字表示弧上的单位费用;节点上的数字表示供需量).

20. 用松弛算法计算练习题 11 网络中的最小费用流.

21. 在如图 6.17 所示的网络中,找出所有的基本解(基本流)及对应的树,并指出哪些是可行的,哪些是强可行的.假设节点上的数字表示供需量,所有弧上的流量只有非负限制,费用任意.

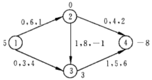


图 6.16　第 19 题图

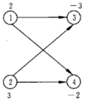


图 6.17　第 21 题图

22. 证明(6.39)和(6.40).

23. 用网络单纯形算法计算练习题 11 网络中的最小费用流.

24. 用网络单纯形算法计算练习题 19 网络中的最小费用流.

25. 考虑用网络单纯形算法计算由一个节点(源)到所有节点的最短路问题:

(1) 写出其最小费用流模型.

(2) 如何使用大-M 法构造初始强可行解?

(3) 证明单纯形算法产生的树弧都是远离源点的,并且含有单位流量.

(4) 如何使用单纯形算法检测负圈的存在?

(5) 假设弧长非负，证明在某种具体实现方式下，网络单纯形算法等价于 Dijkstra 算法.

26. 考虑用网络单纯形算法计算由源点 1,2,3,4 和汇点 5,6,7,8 组成的指派问题：假设所有源点和所有汇点之间都有弧相连，弧(1,6),(1,7),(2,5),(2,7),(3,5),(3,6),(4,8)上的费用都是 1,其他弧上的费用都是 0. 假设选定初始的可行树为{(1,5),(1,6),(2,6),(2,8),(3,7),(4,7),(4,8)},初始解为：(1,5),(2,6),(3,7),(4,8)上的流量为 1,其他弧上的流量为 0. 请你通过这个例子说明：如果不限定出基弧的选择方法(即不一定保持强可行树结构),则算法可能会出现循环.

第 7 章　匹 配 问 题

我们先看如下一个简单例子：在一次有 m 个大学毕业生和 n 家公司参加的供需见面会上，每个毕业生愿意加入到若干家公司中的一家工作，而每个公司愿意接收若干毕业生中的一人到公司工作. 那么，最后最多有多少人可以在这次供需见面会上找到工作(即最多有多少家公司可以在这次供需见面会上招聘到员工)？ 如果每个毕业生到每一家公司工作将会产生的效益不同，那么，为了使得最后产生的总效益最大，最多有多少人可以在这次供需见面会上找到工作？

上面的这类问题就是匹配问题(matching problem)的例子. 匹配问题的研究内容是非常丰富的，本章主要介绍匹配问题的几个基本算法.

7.1　匹配问题的数学描述

定义 7.1　给定一个网络 $N=(V, A, C)$，其匹配(matching)M 定义为弧集合 A 的一个子集，使得 M 中任何两条弧没有公共端点. 在 N 中计算弧数最多的匹配的问题，称为基数匹配问题(cardinality matching problem)，简称匹配问题；在 N 中计算权最大的匹配的问题，称为赋权匹配问题(weighted matching problem).

显然，基数匹配问题是赋权匹配问题的一个特例，即所有弧上的权均为 1 的情形. 前面的例子中，如果把 m 个大学毕业生和 n 家公司的集合作为节点集合，在一名毕业生与一家公司间的弧表示该毕业生愿意加盟该公司、该公司也愿意接受该毕业生，则构成一个二部图网络，例子中的前一个问题是基数匹配问题，后一个问题是赋权匹配问题.

匹配问题中的网络(图)$N=(V, A, C)$一般假设是无向简单网络. 利用整数线性规划模型，令变量 $x_{ij}=1$ 表示弧$(i, j)\in A$ 属于匹配M，而 $x_{ij}=0$ 表示(i, j)不属于匹配 M，则最大权匹配问题可以描述为：

$$\max \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{7.1}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} \leqslant 1, \quad \forall i \in V, \tag{7.2}$$

$$x_{ij} \in \{0, 1\}, \quad \forall (i, j) \in A. \tag{7.3}$$

记无向图的关联矩阵为 $\boldsymbol{B}$，则约束条件(7.2)可以等价地写成

$$\boldsymbol{B}\boldsymbol{x}\leqslant \boldsymbol{e},$$

这里的 $\boldsymbol{e}$ 表示分量全部为 1 的向量. 约束条件(7.2)表示在匹配 M 中每个节点最多只能与一条弧关联，同时也可以保证非负整数 $x_{ij}\leqslant 1$，因此约束条件(7.3)可以替代为

$$x_{ij} \in \mathbb{N}, \quad \forall (i, j) \in A. \tag{7.4}$$

匹配问题在实际中是经常遇到的，但其求解的难易程度与网络(图)$N=(V, A, C)$是否为二部图有很大的关系，二部图上的匹配问题相对比较简单．本章我们将简要介绍4类匹配问题的算法：二部基数匹配问题、二部赋权匹配问题、非二部基数匹配问题、非二部赋权匹配问题(bipartite/nonbipartite cardinality/weighted matching problem)．前面的例子中的匹配问题是二部图的匹配问题．

对于无向二部图的匹配问题，由于关联矩阵$\boldsymbol{B}$为全幺模矩阵，所以一定存在整数最优解(实际上，所有基本可行解都是整数解)，因此约束条件(7.3)和约束条件(7.4)可以进一步替代为

$$x_{ij} \geqslant 0, \quad \forall (i, j) \in A. \tag{7.5}$$

也就是说，此时的匹配问题实际上完全是一个线性规划问题．

对于无向非二部图的匹配问题，关联矩阵$\boldsymbol{B}$一般不是全幺模矩阵，所以x_{ij}的整数约束一般不能直接去掉．我们知道，一个无向图为二部图当且仅当它不含奇圈，因此为了得到非二部匹配问题的线性规划模型，必须消除奇圈带来的影响，即附加一些约束．N中奇数个(大于等于3个)节点组成的子集称为节点奇集(odd-set)，设S_k是N中$2s_k+1$ ($s_k \in \mathbb{N}$)个节点组成的节点奇集，记$T(S_k)=\{(i, j) \in A \mid i, j \in S_k\}$．由于一个长为$2s_k+1$的奇圈最多包含一个匹配$M$的$s_k$条弧，所以

$$\sum_{(i, j) \in T(S_k)} x_{ij} \leqslant s_k, \quad \forall \text{ 节点奇集 } S_k. \tag{7.6}$$

可以证明(比较烦琐，这里不再给出证明)，当约束条件(7.6)加入以后，对x_{ij}的整数约束就可以去掉．所以，对于一般的匹配问题，可以用线性规划模型(7.1)，(7.2)，(7.5)，(7.6)描述．

以上我们给出了匹配问题的线性规划模型，但在实际计算中一般没有必要直接调用线性规划算法(如单纯形算法)进行求解，而是根据匹配问题本身的特点，设计一些高效率的算法．本章只是简要地介绍一些简单的匹配算法，而对于复杂的算法以及有些算法在计算机上的实现细节就不介绍了．为此，本节先介绍一些有关的概念、引理和定理．

定义 7.2 设M是网络$N=(V, A, C)$中的一个匹配，则M中的弧称为匹配弧(matched arc/edge)，$A \backslash M$中弧称为自由弧(free arc/edge)或非匹配弧．由匹配弧相连的两个节点互称配偶(mate)，与匹配弧关联的节点称为匹配点(matched vertex)或已盖点，不与匹配弧关联的节点称为暴露点(exposed vertex)或未盖点(或非匹配点)．如果一个图中的所有节点最多只有一个不是匹配点，则称相应的匹配为完美匹配(perfect matching)．

一般情况下，并不是每个网络都存在完美匹配．显然，如果一个网络存在完美匹配，则完美匹配一定是最大(基数)匹配．

定义 7.3 对于N中的一条起点为暴露点i_0和终点为i_r的路$P=i_0, i_1, \cdots, i_r$，如果

路上的弧由匹配弧和自由弧交替出现，则称 P 为交错路(alternating path)，并称交错路上与起点的距离(即交错路上从起点到该节点的子路所包含的弧的条数)为偶数的节点为外点(outer vertex)，与起点的距离为奇数的节点为内点(inner vertex). 对于交错路 $P=i_0, i_1, \cdots, i_r$，如果 (i_r, i_0) 也属于 A，且 (i_{r-1}, i_r) 是匹配弧而 (i_r, i_0) 为自由弧，则 $P\cup\{(i_r, i_0)\}$ 称为交错圈(alternating cycle). 对于交错路 $P=i_0, i_1, \cdots, i_r$，如果 i_0，i_r 都是暴露点，则称 P 为增广路(augmenting path).

显然，增广路 $P=i_0, i_1, \cdots, i_r$ 的长度一定是奇数，即 i_r 是内点（注意：i_0 永远是外点）；且 P 上除了 i_0，i_r 是暴露点之外，所有其他节点都是匹配点. 此外，交错圈也一定是奇圈(即长度为奇数的圈)，且圈上自由弧的条数比匹配弧的条数多 1. 例如，在图 7.1(a) 所示的网络中，$M=\{(2, 4), (3, 5)\}$ 是一个基数为 2 的匹配(我们在本章中总是用粗线表示匹配弧)，节点 1，6，7 为暴露点，节点 2，3，4，5 为匹配点；圈 1-2-4-5-3-1 是一个交错圈，而 1-2-4-3-5 和 1-2-4-3-5-6 都是交错路，其中 1-2-4-3-5-6 为增广路. 如果我们把 1-2-4-3-5-6 中的匹配弧和自由弧互换，则可以得到一个基数更大(为 3)的匹配(如图 7.1(b)所示). 下面的引理说明，这一结论具有一般性.

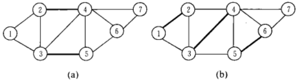


图 7.1　匹配与增广路

引理 7.1　设 P 是网络 N 中关于匹配 M 的增广路，则 M 与 P 的对称差 $M'=M\oplus P=(M\cup P)\setminus(P\cap M)$ 也是匹配，且 $|M'|=|M|+1$.

证明　首先证明 M' 是一个匹配，即 M' 中任何两条弧没有相同的端点. 采用反证法，假设 M' 中存在两条弧 e，e' 有相同的端点，则由对称差的定义可知这两条弧中必然有一条属于 M 而不属于 P，另一条属于 P 而不属于 M. 不妨记 $e\in M\setminus P$，$e'\in P\setminus M$，则 P 不是关于 M 的增广路，矛盾. 所以 M' 是一个匹配.

由于 P 是关于 M 的增广路，所以 $|P|=2|P\cap M|+1$. 因此 $|M'|=|(M\cup P)\setminus(P\cap M)|=|M|+|P|-2|P\cap M|=|M|+1$. □

上述操作 $M'=M\oplus P$ 称为一次增广(augmentation). 显然，经过一次增广，原来的匹配点仍然保持为匹配点，且原来的两个暴露点(即 P 的两个端点)也转变为匹配点. 类似于最大流的讨论，我们可以得到如下增广路定理.

定理 7.1　网络 $N=(V, A, C)$ 中的匹配 M 为最大匹配，当且仅当 N 中不存在关于匹配 M 的增广路.

证明　必要性可以由引理 7.1 直接得到. 下面证明充分性. 假设 N 中不存在关于匹配

M 的增广路，且 M 不是最大匹配，即存在匹配 M'，使得 $|M'|>|M|$. 考虑子图 $G'=(V, M\oplus M')$，则 G'中每个节点的度数小于等于 2. 如果 G'中某节点的度数等于 2，则它所关联的两条弧必然分属于 M 和 M'. 于是，虽然 G'不一定连通，但 G'的每一个连通子图或者是一条路，或者是偶数长的圈，且这些路(圈）关于 M 和 M'都一定是交错路(圈). 对于每个圈，它所包含的 M'中的弧数等于它所包含的 M 中的弧数. 因此，一定存在一条路，它所包含的 M'中的弧数大于它所包含的 M 中的弧数，所以它是关于匹配 M 的增广路，矛盾. □

类似于最大流增广路算法的讨论，根据定理 7.1，为了得到最大匹配，我们可以从任何匹配 M 开始，如果当前匹配 M 不是最大匹配，我们从其中一个暴露点出发，寻找增广路并增广，直到找不到增广路为止. 所以，问题的关键是如何有效地寻找增广路. 为了后面讨论算法的方便，下面我们再介绍一个与增广路相关的更一般的结论.

定理 7.2　已知网络 $N=(V, A, C)$中的匹配 M 和暴露点 p，如果网络中不存在以 p 为起点的增广路，则 N 中存在最优匹配 M^*，使得 p 关于 M^* 也是暴露点.

证明　设 M'为最优匹配，若 p 关于M'是暴露点，则定理成立. 若 p 关于 M'是匹配点，考虑子图 $G'=(V, M\oplus M')$. 显然，G'的每一个连通子图或者是一条路，或者是偶数长的圈，且这些路(圈)关于 M 和 M'都一定是交错路(圈). 对于每个圈，它所包含的每个节点关于 M 和 M'都一定是匹配点. 由于 p 关于 M'是匹配点而关于 M 是暴露点，所以在 G'中 p 一定是一条交错路 P 的起点；又由于网络中不存在以 p 为起点的关于 M 的增广路，所以该交错路 P 的长度一定是偶数. 令 $M^*=M'\oplus P$，则 $|M^*|=|M'|$，即 M^* 也是最大匹配，且 p 关于 M^* 也是暴露点. □

7.2　二部基数匹配问题

7.2.1　增广路算法

对于基数匹配问题，由于与网络上的权无关，因此我们直接将网络写成 $N=(V, A)$. 如果当前匹配 M 不是最大匹配，我们从一个暴露点出发，寻找增广路并进行增广. 根据定理 7.2，如果从某个暴露点 p 出发找不到增广路，则可以把该节点 p 以及它所关联的弧从 N 中删去，而不会影响最大匹配的基数.

为了查找出从暴露点 p 出发的一条增广路，我们需要有一个系统的方法有效地检查从 p 出发的交错路. 也就是说，我们希望遍历从 p 出发沿交错路可以到达的所有节点；一旦某个这类节点是一个暴露点，则找到了一条增广路. 前面说过，交错路上总是外点和内点交替出现，自由弧和匹配弧交替出现. 在遍历时，对于一个外点，我们需要考虑所有与它相邻的节点；而对于一个内点，我们只需要考虑与它匹配的配偶节点. 为了区分外点和内点，以及一个节点是否已经被检查过，可以利用标号的思想. 具体算法如下：

二部基数匹配问题的增广路算法

STEP0 在二部图网络 $N=(V_1, V_2, A)$ 中任取一个匹配 M(如 $M=\varnothing$)，置所有 N 中节点 i 的标号为 label(i)＝'NULL'(这里'NULL'表示没有标号).

STEP1 如果 N 中不存在暴露点，则 M 为最大匹配，结束. 否则选取 N 中一个暴露点 p，置节点 p 的标号为 label(p)＝'OUTER'，LIST＝$\{p\}$.

STEP2 如果 LIST＝$\varnothing$，则表明不存在从 p 出发的增广路，所以将节点 p 以及它所关联的弧从 N 中删去，转 STEP1. 否则从 LIST 中取出一个节点 i，令 LIST＝LIST$\setminus\{i\}$. 如果 label(i)＝'OUTER'，转 STEP3；否则转 STEP4.

STEP3 对 i 的每个相邻节点 j：如果 j 为暴露点，令 $q=j$，pred(q)＝i，转 STEP5；如果 j 为匹配点且 label(j)＝'NULL'，令 pred(j)＝i，label(j)＝'INNER'，LIST＝LIST$\cup\{j\}$. 转 STEP2.

STEP4 记 i 的匹配点为 j：如果 label(j)＝'NULL'(无标号)，令 pred(j)＝i，label(j)＝'OUTER'，LIST＝LIST$\cup\{j\}$，转 STEP2；否则直接转 STEP2.

STEP5 利用数组 pred 所记录的信息，从节点 q 反向追踪确定一条 p-q 增广路 P，执行增广操作 $M=M\oplus P$. 置所有 N 中节点 i 的标号为 label(i)＝'NULL'，转 STEP1.

上述算法希望查找的从暴露点 p 出发的所有交错路构成一棵以 p 为根的树，称为交错树(alternating tree). 上述算法希望遍历交错树而找到从暴露点 p 出发的一条增广路. 如果找到了这样的增广路，则进行增广，此后该节点永远保持为匹配点，这自然没有问题. 如果算法没有找到这样的增广路，我们是否可以断定一定不存在这样的增广路? 对于一般图的匹配问题，这一点是不一定成立的. 也就是说，即使存在从暴露点 p 出发的增广路，上述算法也不一定能发现它，即能否发现增广路与算法检查节点的顺序有关系. 所以，上面的算法应用于求解一般图的匹配问题可能得不到最大匹配. 然而，对于二部图的匹配问题，这一点却是一定成立的，即只要存在从暴露点 p 出发的增广路，上述算法就一定能发现它. 这是因为，对于二部图，一定存在节点 V 的划分 V_1 和 V_2，使得 A 中的弧总是横跨在 V_1 和 V_2 之间，所以当 $p\in V_1$ 时，所有外点都属于 V_1，所有内点都属于 V_2；$p\in V_2$ 时，所有外点都属于 V_2，所有内点都属于 V_1. 也就是说，无论以何种顺序检查节点，所有节点最后获得的标号("内点(INNER)"或"外点(OUTER)")都是唯一的. 因此，只要存在一条增广路，即只要有一个内点为暴露点，算法一定能分辨该暴露点为内点(标号'INNER')，即一定找到一条增广路. 所以，上面的算法一定得到二部图的最大匹配.

例 7.1 在图 7.2(a)所示的网络中，计算最大匹配(假设一个当前匹配已经知道，已经标示在图中).

假设 LIST 中的节点按照先进先出的顺序检查，且每次检查一个节点时对它的邻居按照节点标号由小到大的顺序检查. 我们把各节点的标号 label 和 pred 直接表示在节点旁边

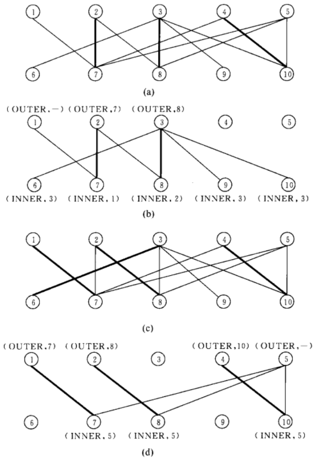


图 7.2　二部图匹配的例子(例 7.1)

的括号中（第 1 部分表示标号 label，第 2 部分表示 pred，即从何处获得标号）. 从当前匹配开始，计算过程如下：

1. 图中有 4 个暴露点{1，5，6，9}，我们首先从节点 1 开始，即 LIST＝{1}. 依次检查节点 1，7，2，8，3，此时得到标号节点组成的交错树如图7.2(b). 此时 LIST＝{6，9，10}，继续检查节点 6 发现它是一个暴露的内点，则表明找到从 1 出发的增广路 1-7-2-8-3-6，所以进行增广得到图 7.2(c).

2. 去掉所有节点的标号. 图中有两个暴露点{5，9}，我们从节点 5 开始，即 LIST＝{5}. 检查节点 5 得到 LIST＝{7，8，10}. 依次检查节点 7，8，10，1，2，4，得到标号节点组成的交错树如图 7.2(d). 此时 LIST＝∅，则表明不存在从 5 出发的增广路，所以将节点 5

以及它所关联的弧(5, 7), (5, 8), (5, 10)从 N 中删去. 此时只有一个暴露节点 9(第 1 排没有暴露点), 已经得到最优解 $M=\{(1, 7), (2, 8), (3, 6), (4, 10)\}$. □

算法复杂度分析: 由于每次针对暴露点 p 的迭代中, 如果找到了一条 p-q 增广路, 则进行增广, 此后节点 p, q 永远保持为匹配点; 如果算法没有找到增广路, 我们删去 p 及其关联的弧, 所以算法最多执行 $O(n)$ 次遍历搜索和增广. 一次增广(STEP5)最多需要 $O(n)$ 次基本操作. 选取 N 中一个暴露点 p 以后, 对于每个外点, 我们最多需要考虑所有与它相邻的节点; 而对于一个内点, 我们只需要考虑与它匹配的配偶节点. 所以, 选取 N 中一个暴露点 p 以后, 遍历搜索的复杂度为 $O(m)$. 因此, 上述增广路算法的总的时间复杂度为 $O(mn)$.

7.2.2 应用简单网络上的最大流算法

对于二部图, 一定存在 V 的划分 V_1 和 V_2, 使得 A 中的弧总是横跨在 V_1 和 V_2 之间, 因此直接将网络写成 $N=(V_1, V_2, A)$. 我们首先把横跨在 V_1 和 V_2 之间的无向弧(i, j)修改为有向弧(i, j); 然后增加两个节点 s, t, 并增加弧(s, i)和(j, t). 记新的网络为 N', 令所有弧上的容量为 1, 则最大匹配问题转化为 N'的最大流问题. 这是因为, 对于原网络 N 中任意一个匹配 M, 一定存在 N'中的一个整可行流 $\boldsymbol{x}$ 与之对应; 反之, 对于 N'中的一个整可行流 $\boldsymbol{x}$, 一定存在 N 中的一个匹配 M 与之对应; 并且有 $|M|=v(\boldsymbol{x})$.

回顾第 5.6.2 节可以知道, N'是一个单位容量简单网络. 因此, 应用该节介绍的最大流算法, 求解二部匹配问题的复杂度为 $O(n^{1/2}m)$. 这是目前求解二部匹配问题的复杂度最低的算法.

7.3 非二部基数匹配问题

我们这一节将二部图的增广路算法推广到一般图. 7.2 节介绍的增广路算法应用于求解一般图的匹配问题可能得不到最大匹配, 也就是说, 即使存在增广路, 上述算法也不一定能发现它, 即能否发现它与算法检查节点的顺序有关系. 本节首先介绍"花"的概念.

例 7.2 我们再通过一个例子观察一下上述算法为什么不一定能发现所有可能的增广路. 在如图 7.3 所示的网络中, 存在从暴露点 $p=1$ 出发的增广路 1-2-3-6-7-8. 如果从暴露点 $p=1$ 出发搜索增广路时, 对圈上的节点以反时针方向标号, 则节点 1, 3, 7 被标号为外点("OUTER"), 节点 2, 6, 8 被标号为内点("INNER"), 因此可以找到该增广路. 如果对圈上的节点以顺时针方向标号, 则节点 1, 3, 5 被标号为外点("OUTER"), 节点 2, 4, 7 被标号为内

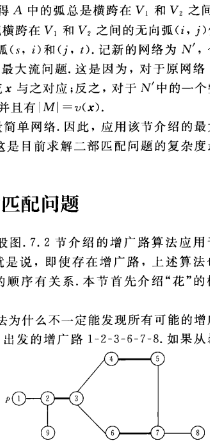


图 7.3 增广路算法失败的例子(例 7.2)

点("INNER").此时由于节点7为内点，因此下一个得到标号点只有7的配偶6（节点6被标号为外点），最后找不到增广路. □

通过这个例子，我们可以看出问题就出在网络中出现了奇圈（图中的3-4-5-7-6-3），使得沿着圈的不同方向标号时，节点得到的标号不一样.也就是说，圈上的所有节点既可能是内点，也可能是外点.我们在7.2节介绍的算法中，由于只有当节点为外点时才对所有与之相邻的节点进行标号，而当节点为内点时只对与之匹配的节点进行标号，所以可能找不到所有增广路.如果我们使得奇圈上的节点一定可以得到外点标号，则应该可以找到所有增广路.这就是把二部图的匹配算法推广到一般图上去的关键所在.

回顾7.2节的算法，我们知道算法不会对已经得到标号的节点重新标号，因此奇圈上的节点只可能得到"内点"或"外点"标号中的一个，而不可能保证都一定得到外点标号.我们可能马上想到一种可能的改进，即在算法过程中允许对已经得到标号的节点重新标号.但实际上，这一做法不是一个有效的方法.例如，在上面的例子中，如果沿1-2-3-6-7-5-4-3-2-9标号，则节点2和节点3在标号后又得到新的标号，最后节点9得到内点标号.此时，算法将报告发现了一条从节点1至节点9的增广路，然而实际上网络中并不存在这样的增广路.因此，我们必须从其他方面考虑对7.2节的算法进行改进，通常采用的办法是下面将要介绍的所谓"花朵收缩"技巧.为此我们先介绍"花"的概念.

定义7.4 对于网络N的匹配M和暴露点p，由满足以下条件的一条交错路和一个奇圈所构成的子图称为"花"(flower)：

(1) 该子图中交错路是从p到q的一条交错路，且q在该交错路上是一个外点；

(2) 该子图中奇圈与交错路的唯一交点为q，圈上自由弧与匹配弧交替出现，且圈上与q关联的两条弧都是自由弧；

此时，称p-q交错路为花茎(stem)，称q-q奇圈为花朵(blossom)，而称p为根(root)，称q为花蒂(base).

例如，在上面的例子中，子图1-2-3-6-7-5-4-3为花，路1-2-3为花茎，圈3-6-7-5-4-3为花朵，节点$p=1$为根，节点$q=3$为花蒂.一般情况下，我们也允许p和q重合，此时花茎为空集.

显然，花茎的长度一定是偶数，且从根p出发在花上不存在增广路.为了找到增广路，一定要至少增加一条不在花上的弧.虽然花朵上的节点可以是内点也可以是外点，但只有当它是外点时才可能对与之相邻的不在花上的节点进行标号，所以我们希望使得花朵上的节点一定可以得到外点标号.通常采用的办法是把花朵收缩(contract或shrink)成为一个节点进行处理，当找到一条增广路以后再将它展开.这一技巧与我们在3.3节和3.4节中所采用的方法有许多类似之处.

定义7.5 对于网络N的匹配M和花朵W，网络N关于W的收缩(记为N/W)定义为：

(1) 用人工节点w代替花朵W，即增加人工节点w而删去花朵W中的所有节点；

(2) 修改原网络中的弧，即所有原来与 W 中节点相邻的节点变成与 w 相邻.

那么，我们如何才能知道网络中存在花，并找到花朵进行收缩呢？我们可以仍然按照7.2节介绍的算法进行搜索，但是在算法过程中增加检查是否遇到对已经标号过的节点可能赋予不同的新标号的情况. 当网络中存在花时，一定会遇到对已经标号过的节点 i 可能赋予不同的新标号的情况，此时我们并不对该节点 i 的标号真的进行修改，而仅仅报告发现网络中存在花，因此需要找到对应的花朵并对它进行收缩. 为了找到对应的花朵，我们可以同时从当前节点和节点 i 出发沿交错路反向追踪，找到两条交错路上的第一个公共点 w，则找到了对应的花朵，其中 w 为花蒂. 具体算法如下：

非二部图的匹配算法

STEP0　在网络 N 中任取一个匹配 M(如 $M=\varnothing$)，置所有 N 中节点 i 的标号为 label(i)='NULL'(这里'NULL'表示没有标号).

STEP1　如果 N 中不存在暴露点，则 M 为最大匹配，结束. 否则选取 N 中一个暴露点 p，置节点 p 的标号为 label(p)='OUTER'，LIST=$\{p\}$.

STEP2　如果 LIST=$\varnothing$，则表明不存在从 p 出发的增广路，所以将节点 p 以及它所关联的弧从 N 中删去，转 STEP1. 否则从 LIST 中取出一个节点 i，令 LIST=LIST$\setminus\{i\}$. 如果 label(i)='OUTER'，转 STEP3；否则转 STEP4.

STEP3　对 i 的每个相邻节点 j：如果 label(j)= 'OUTER'，转 STEP5；如果 j 为暴露点，令 $q=j$，pred(q)=i，转 STEP6；如果 j 为匹配点且 label(j)='NULL'，令 pred(j)=i，label(j)='INNER'，LIST=LIST$\cup\{j\}$. 转 STEP2.

STEP4　记 i 的匹配点为 j：如果 label(j)='INNER'，转 STEP5；如果 label(j)='NULL'(无标号)，令 pred(j)=i，label(j) ='OUTER'，LIST=LIST$\cup\{j\}$，转 STEP2；否则直接转 STEP2.

STEP5　找到对应的花朵，并将该花朵收缩为一个人工节点，同时修改网络和 LIST. 令该人工节点的标号为 'OUTER'，并将该人工节点加入 LIST 中，转STEP2.

STEP6　利用数组 pred 所记录的信息，从节点 q 反向追踪确定一条 p-q 增广路 P'. 展开 P' 上的人工节点，确定一条不包括人工节点的增广路 P. 执行增广操作 $M=M\oplus P$. 置所有 N 中节点 i 的标号为 label(i)='NULL'，转 STEP1.

例 7.3　在图 7.4(a)所示的网络中，计算最大匹配(假设一个当前匹配已经知道，已经标示在图中).

假设每次检查一个节点时对它的邻居按照节点标号由小到大的顺序检查. 但是，为了说明算法时的方便，我们在这个例子里对 LIST 中的节点的检查顺序并不特别指定. 我们把各节点的标号 label 和 pred 直接表示在节点旁边的括号中(第 1 部分表示标号 label，第 2

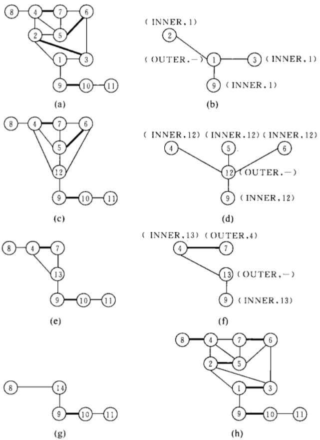


图 7.4 非二部图匹配的例子(例 7.3)

部分表示 pred，即从何处获得标号).

从当前匹配开始，计算过程如下：

(1) 图中有 3 个暴露点(1, 8, 11)，我们首先从节点 1 开始，即 LIST＝{1}. 取出节点 1 进行检查，得到交错树如图 7.4(b). 此时 LIST＝{2, 3, 9}，取出节点 2 进行检查，发现其配偶节点 3 也具有内点标号，所以发现了一个花朵 $S_1=\{1, 2, 3\}$. 收缩 S_1 为人工节点 12，则网络如图 7.4(c).

(2) 此时 LIST＝{9, 12}，我们优先选取人工节点 12，继续标号，得到的交错树如图 7.4(d). 此时 LIST＝{4, 5, 6, 9}，假设选取节点 4 进行检查，则其配偶节点 7 得到标号. 继续选取节点 5 进行检查，发现其配偶节点 6 也具有内点标号，所以发现了一个花朵 $S_2=\{12, 5, 6\}$. 收缩 S_2 为人工节点 13，则网络如图 7.4(e).

(3) 此时 LIST={7, 9, 13}, 交错树如图 7.4(f). 我们优先选取人工节点 13, 继续标号, 发现其相邻节点 7 也具有外点标号, 所以发现了一个花朵 $S_3=\{13, 4, 7\}$. 收缩 S_3 为人工节点 14, 则网络如图 7.4(g).

(4) 此时 LIST={9, 14}, 取出节点 14 进行检查, 发现其相邻节点 8 为暴露点, 因此找到了一条增广路 14-8.

(5) 依次展开节点 14, 13, 12, 得到增广路 14-8 即为 13-7-4-8, 或等价地为 12-5-6-7-4-8, 即 1-3-2-5-6-7-4-8. 增广得到图 7.4(h).

(6) 此时图中只有一个暴露节点 11, 继续搜索不可能找到增广路, 因此已经得到最优解. □

算法复杂度分析: 我们这里只是粗略地介绍一下上述算法的复杂性. 首先, 与二部匹配算法类似, 非二部匹配算法最多执行 $O(n)$ 次遍历搜索和增广. 但由于搜索中可能有收缩, 增广中可能要展开人工节点, 因此一般来说总的复杂度高于二部匹配算法. 但可以注意到, 每个花朵至少含有 3 个节点, 收缩后变成一个人工节点, 因此每次收缩至少使得节点数减少两个. 所以, 每次搜索到一条增广路所执行的收缩次数最多不超过 $n/2$ 次(自然, 每次增广中展开人工节点的次数也最多不超过 $n/2$ 次). 通过细致的实施, 可以使得这最多 $n/2$ 次收缩和展开的总复杂度为 $O(n^2)$. 因此, 上述增广路算法的总的时间复杂度为 $O(n^3)$.

从历史上看, 非二部匹配算法是由 Edmonds 于 1965 年首先提出的, 当时给出的算法的复杂度为 $O(n^4)$. 此后, 许多研究将算法的复杂度不断改进为 $O(mn^2)$, $O(n^3)$, $O(n^{5/2})$, $O(mn)$ 等. 1980 年 Micali 和 Vazirani 将算法的复杂度改进为 $O(mn^{1/2})$, 这是目前复杂度最好的算法. 而且可以看出, 这一结果与 7.2 节利用单位容量简单网络所得到的二部匹配算法的复杂度是一样的. 从这个角度来看, 二部匹配与非二部匹配在算法的复杂度上没有本质的区别.

7.4　二部赋权匹配问题

对于赋权匹配问题, 我们不妨假设所有弧上的权非负, 因为负权弧一定不会出现在最大匹配中. 也就是说, 如果某条弧上的权为负数, 我们可以去掉这条弧, 或者将这条弧上的权修改为 0.

对于二部图, 一定存在节点集 V 的划分 V_1 和 V_2, 使得图中的弧总是横跨在 V_1 和 V_2 之间, 因此可以直接将网络写成 $N=(V_1, V_2, A, C)$. 我们可以不妨假设 $|V_1|=|V_2|$, 因为当某个集合中的节点数少于另外一个集合中的节点数时, 我们总可以在节点数较少的集合中增加节点, 使得 $|V_1|=|V_2|$. 进一步, 我们可以不妨假设二部图为完全图, 因为如果不是完全图, 我们可以增加弧使它变成完全图, 并令这些增加的弧上的权为 0. 显然, 这些操作不会影响最大权匹配的最优值.

也就是说, 我们只需要考虑非负权的完全二部图网络 N 上的匹配问题. 在这种情况

下，我们可以证明一定存在一个完美匹配为最大权匹配. 这是因为，如果某个最大权匹配不是完美匹配，而由于二部图为完全图，所以总可以在该最大权匹配中增加一些弧使得它变成完美匹配. 注意到所有弧上的权非负，这样做不会减少匹配的权. 在这样的假设下，下面我们将说明二部赋权匹配问题可以很容易地转化为最小费用流问题.

记$|V_1|=|V_2|=n$. 我们首先把横跨在V_1和V_2之间的无向弧(i, j)修改为有向弧(i, j)，令弧上的容量为1，费用$c'_{ij}=K-c_{ij}$(K为一个充分大的正整数，保证c'_{ij}为非负整数). 然后增加两个节点s, t，并对V_1中的所有节点i和V_2中的所有节点j增加有向弧(s, i)和(j, t)，令所有这些弧上的容量为1，费用$c'_{ij}=0$. 记新的网络为N'，则最大赋权匹配问题转化为N'的最小费用流问题. 这是因为，对于原网络N中任意一个匹配M，一定存在N'中的一个整可行流$\boldsymbol{x}$与之对应；反之，对于N'中的一个整可行流$\boldsymbol{x}$，一定存在N中的一个匹配M与之对应；并且我们一定有$|M|=v(\boldsymbol{x})$, $C'(M)=K|M|-C(M)$. 由于对于N中的任意一个完美匹配M，一定有$|M|=n$，所以我们可以在N'中计算一个流值为n的最小费用整可行流，它所对应的完美匹配一定就是最大权匹配.

第6章中介绍的算法都可以用来求解本节的匹配问题. 这里我们只是对其中的两种算法作一点简单的分析.

假设采用最短增广路算法来求解本节的匹配问题. 第6章中我们分析过该算法求解流值为v的最小费用流问题的复杂度为$O(vS(n, m, nC))$，其中$S(n, m, C)$为非负权网络的最短路算法的复杂度. 之所以$S(n, m, nC)$中第3个参数为nC，是因为残量网络中弧上的费用上界为nC. 对于本节的匹配问题，由于最短增广路长度固定为3，所以该算法的复杂度可以表示为$O(nS(n, m, C))$.

假设采用原始-对偶算法来求解本节的匹配问题. 记求解非负弧长网络的最短路算法的复杂度为$S(n, m, C)$，最大流算法的复杂度为$M(n, m, U)$，第6章中我们分析过一般情况下本算法复杂度为$O(\min\{nU, nC\}[S(n, m, nC)+M(n, m, U)])$. 对于本节的匹配问题，由于增广路的特殊性，所以即使采用Ford-Fulkerson标号算法，计算最大流的复杂度也不会超过$O(nm)$. 因此，原始-对偶算法来求解本节的匹配问题的复杂度也可以表示为$O(nS(n, m, C))$.

在运筹学中，一般把二部赋权匹配问题称为指派问题，而把上面的原始-对偶算法称为匈牙利算法. 由于目前求解非负权网络的最短路问题的强多项式算法的复杂度最好为$O(m+n\log n)$，所以目前求解二部赋权匹配问题的复杂度最低的强多项式算法的复杂度最好为$O(mn+n^2\log n)$. 对于完全图，由于$m=O(n^2)$，所以复杂度为$O(n^3)$.

7.5 非二部赋权匹配问题

对于赋权匹配问题，7.4节我们说过可以假设所有弧上的权非负. 进一步，我们可以不妨假设网络图为完全图且节点个数为偶数，因为否则的话，我们可以增加一个节点和若干

弧使它变成偶数阶完全图，并令这些增加的弧上的权为0.显然，这些操作不会影响最大权匹配的最优值.

对于一般的赋权匹配问题，我们在7.1节中建立了它的线性规划模型(7.1)，(7.2)，(7.5)，(7.6).通过约束条件(7.2)和(7.5)引入对偶变量$\boldsymbol{\alpha}$和$\boldsymbol{\beta}$，可以得到该线性规划模型的对偶问题为

$$\min\left(\sum_{i\in V}\alpha_i+\sum_{S_k\in OS}s_k\beta_k\right) \tag{7.7}$$

$$\text{s.t.}\ \ \alpha_i+\alpha_j+\sum_{k:(i,j)\in T(S_k)}\beta_k\geqslant c_{ij},\quad \forall(i,j)\in A, \tag{7.8}$$

$$\alpha_i\geqslant 0,\quad \forall i\in V, \tag{7.9}$$

$$\beta_k\geqslant 0,\quad \forall S_k\in OS. \tag{7.10}$$

其中OS表示网络中所有节点奇集的集合.当网络中的节点数n为偶数时，

$$|OS|=\binom{n}{3}+\binom{n}{5}+\cdots+\binom{n}{n-1}=2^{n-1}-n. \tag{7.11}$$

记$\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$，令

$$c_{ij}^{\pi}=\alpha_i+\alpha_j+\sum_{k:(i,j)\in T(S_k)}\beta_k-c_{ij},\quad \forall(i,j)\in A, \tag{7.12}$$

则互补松弛条件为

$$c_{ij}^{\pi}x_{ij}=0,\quad \forall(i,j)\in A, \tag{7.13}$$

$$\left(\sum_{(i,j)\in T(S_k)}x_{ij}-s_k\right)\beta_k=0,\forall\ \text{节点奇集}\ S_k\in OS(|S_k|=2s_k+1), \tag{7.14}$$

$$\left(\sum_{j:(i,j)\in A}x_{ij}-1\right)\alpha_i=0,\quad \forall i\in V. \tag{7.15}$$

下面，我们采用原始-对偶算法进行求解.原始-对偶算法的基本思想是，算法在执行过程中永远保持$\boldsymbol{x}$的原始可行性和$\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$的对偶可行性，并保持式(7.13)和式(7.14)成立.通过迭代改进，当(7.15)也成立时，则已经得到了最优解.

为了永远保持(7.13)，我们看到$\boldsymbol{x}$只有在下面的弧集合上才可以为非零(即匹配弧一定位于下面的弧集合中)：

$$A'=\{(i,j)\in A\mid c_{ij}^{\pi}=0\}. \tag{7.16}$$

为了永远保持(7.14)，我们看到β_k只有在节点奇集S_k为花朵(即S_k中正好含有s_k条匹配弧)时才可以为非零.

而所谓(7.15)成立，也就是说，对于任意一个节点i，要么$\alpha_i=0$,要么i是匹配点.反过来说，如果没有达到最优解，则一定存在使得$\alpha_i>0$的暴露点i.算法的目标就是要通过迭代使得这种不满足(7.15)的节点减少：首先考虑对当前匹配进行增广，使得更多的节点变成匹配点；如果不能找到增广路，则对对偶变量进行修改.

由于匹配弧一定位于A'中，所以为了保持(7.13)，我们只应该在子网络$N'=(V,A')$

中查找增广路. 在前面介绍的计算一般图和二部图的最大基数匹配的增广路算法中，我们可以看出：查找从一个暴露点出发的增广路的过程，实际上是生成一棵以该节点为根的交错树. 如果对多个暴露点出发进行查找，则实际上是生成一个交错森林. 而且，由于展开一个已经收缩的花朵以后，这个花朵可能又会在后续操作中继续出现而再次被收缩，所以我们在不必要时可以暂时不将所有人工节点展开，这样可以避免一些重复计算. 那么，什么样的人工节点不必要展开呢？为了保持(7.14)，当 $\beta_k>0$ 时，节点奇集 S_k 一定是花朵(即 S_k 中一定含有 s_k 条匹配弧)，因此没有必要展开. 所以，我们可以只展开 $\beta_k=0$ 所对应的人工节点.

如果不能找到增广路，则需要对对偶变量进行修改. 这一对对偶变量的修改过程的细节比较复杂，我们将在以下的算法中详细介绍.

初始的原始可行解 $\boldsymbol{x}$ 和对偶可行解 $\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$ 可以取为

$$x_{ij}=0,\quad \forall (i,j)\in A, \tag{7.17}$$

$$\alpha_i=\frac{1}{2}\max_{(j,k)\in A} c_{jk},\quad \forall i\in V, \tag{7.18}$$

$$\beta_k=0,\quad \forall \text{节点奇集 } S_k\in OS. \tag{7.19}$$

容易验证这样定义的为对偶可行解，且满足式(7.13)和式(7.14).

下面给出具体算法如下：

非二部赋权匹配问题的原始-对偶算法

STEP0　在网络 N 中按照(7.17)～(7.19)初始化 $\boldsymbol{x}$ 和 $\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$，此时 $M=\varnothing$. 置所有 N 中节点 i 的标号为 label(i)=‘NULL’(这里‘NULL’表示没有标号).

STEP1　如果 N 中不存在满足 $\alpha_p>0$ 的暴露点 p，则 M 为最大匹配(如果存在人工节点，则需要将人工节点展开，且展开时对应的奇圈是花朵)，结束. 否则继续 STEP2.

STEP2　在 $N'=(V,A')$ 中寻找所有可能的增广路.

STEP2.1　如果 N' 中不存在无标号的暴露点，转 STEP4；否则选取 N' 中一个无标号的暴露点 p，置节点 p 的标号为 label(p)=‘OUTER’，LIST=$\{p\}$.

STEP2.2　如果 LIST=$\varnothing$，则说明 N' 中不存在从 p 出发的增广路，转 STEP4.1；否则从 LIST 中取出一个节点 i，令 LIST=LIST$\setminus\{i\}$. 如果 label(i)=‘OUTER’，转 STEP2.3；否则转 STEP2.4.

STEP2.3　对 i 的每个在 N' 中的相邻节点 j：如果 label(j)=‘OUTER’，转 STEP 2.5；如果 j 为暴露点，令 $q=j$，pred(q)=i，说明找到了增广路，转 STEP2.6；如果 j 为匹配点且 label(j)=‘NULL’，令 pred(j)=i，label(j)=‘INNER’，LIST=LIST$\cup\{j\}$. 转 STEP2.2.

STEP2.4　记 i 的匹配点为 j(j 一定在 N' 中)，如果 label(j)=‘INNER’，转 STEP2.5；如果 label(j)=‘NULL’(无标号)，令 pred(j)=i，label(j)=

'OUTER'，LIST=LIST$\cup\{j\}$，转 STEP2.2；否则直接转 STEP2.2.

STEP2.5　找到对应的花朵，并将该花朵收缩为一个人工节点，同时修改网络 N'和 LIST. 令该人工节点的标号为'OUTER'，并将该人工节点加入 LIST 中，转 STEP2.2.

STEP2.6　利用数组 pred 所记录的信息，从节点 q 反向追踪确定一条 p-q 增广路 P'. 展开 P'上的人工节点，确定一条不包括人工节点的增广路 P. 转 STEP3.

STEP3　增广 M（修改 $\boldsymbol{x}$）：执行增广操作 $M=M\oplus P$. 在 N'中展开所有满足 $\beta_k=0$ 的花朵（节点奇集）S_k，并修改其他相关花朵的花蒂. 去掉所有节点的标号（即标号为'NULL'，令交错森林为空集）. 转 STEP2.

STEP4　修改对偶变量. 具体做法如下：

STEP4.1　定义如下集合：

$OS^+=\{S_k\in OS\mid S_k$ 已经收缩为人工节点，且标号为外点$\}$，

$OS^-=\{S_k\in OS\mid S_k$ 已经收缩为人工节点，且标号为内点$\}$，

$OS^\sim=\{S_k\in OS\mid S_k$ 已经收缩为人工节点，但未被标号$\}$，

$V^+=\{v\in V\mid v$ 被标号为外点或位于 OS^+ 的某个奇集中$\}$，

$V^-=\{v\in V\mid v$ 被标号为内点或位于 OS^- 的某个奇集中$\}$，

$V^\sim=\{v\in V\mid v$ 为无标号点或位于 $OS^\sim$ 的某个奇集中$\}$，

$A^+=\{(i,j)\in A\mid i\in V^+,\ j\notin V^\sim\}$，

$A^{++}=\{(i,j)\in A\mid i\in V^+,\ j\notin V^+$，且不存在 $S_k\in OS^+$ 同时包含 i 和 $j\}$.

STEP4.2　计算

$$\delta_1=\min_{i\in V^+}\alpha_i,$$

$$\delta_2=\begin{cases}\dfrac{1}{2}\min\limits_{S_k\in OS^-}\beta_k, & OS^-\neq\varnothing,\\ \infty, & OS^-=\varnothing,\end{cases}$$

$$\delta_3=\begin{cases}\dfrac{1}{2}\min\limits_{(i,j)\in A^{++}}c_{ij}^{\pi}, & A^{++}\neq\varnothing,\\ \infty, & A^{++}=\varnothing,\end{cases}$$

$$\delta_4=\begin{cases}\min\limits_{(i,j)\in A^+}c_{ij}^{\pi}, & A^+\neq\varnothing,\\ \infty, & A^+=\varnothing,\end{cases}$$

$$\delta=\min\{\delta_1,\delta_2,\delta_3,\delta_4\}.$$

STEP4.3　修改对偶变量如下：

$$\alpha_i=\begin{cases}\alpha_i-\delta, & i\in V^+,\\ \alpha_i+\delta, & i\in V^-,\\ \alpha_i, & \text{其他},\end{cases}$$

$$\beta_k = \begin{cases} \beta_k + 2\delta, & S_k \in OS^+, \\ \beta_k - 2\delta, & S_k \in OS^-, \\ \beta_k, & \text{其他}. \end{cases}$$

STEP4.4 如果 N 中不存在满足 $\alpha_p > 0$ 的暴露点 p，则 M 为最大匹配（如果存在人工节点，则需要将人工节点展开，且展开时对应的奇圈是花朵），结束. 否则继续 STEP4.5.

STEP4.5 展开所有满足 $\beta_k = 0$ 的花朵（节点奇集）S_k，修改子网络 N'. 如果某条弧 (u, v) 被加入 N' 中，且 u 和 v 在交错森林中位于两棵不同的交错子树上，则找到了一条增广路（记该增广路为 P'，在原网络中对应的增广路为 P），转 STEP3. 否则保持当前交错森林不变，转 STEP2.

下面我们需要说明上述算法确实是正确的. 也就是说，我们需要证明算法一定在有限步内终止，且终止时原始可行解 $\boldsymbol{x}$ 和对偶可行解 $\boldsymbol{\pi} = (\boldsymbol{\alpha}, \boldsymbol{\beta})$ 满足式(7.13)～式(7.15). 由于只在 N' 中查找增广路，所以增广操作（STEP3）不会破坏式(7.13). 又由于只展开 $\beta_k = 0$ 所对应的人工节点，所以增广操作（STEP3）也不会破坏式(7.14). 所以，问题的关键在于对偶变量的修改操作也保持式(7.13)和式(7.14)，且在有限步内使得式(7.15)也满足.

引理 7.2 在上述算法对对偶变量的修改操作中，$0 < \delta \leqslant \frac{1}{2} \max\limits_{(j,k)\in A} c_{jk}$.

证明 对 $\delta_1, \delta_2, \delta_3, \delta_4$ 的取值分别讨论.

(1) 我们首先证明 $\delta_1 > 0$. 在修改对偶变量之前，一定存在至少一个暴露点 $i \in V^+$ 满足 $\alpha_i > 0$，因为否则的话算法已经得到了最优解，无需修改对偶变量. 此外，在算法运行到目前为止，暴露点 i 应当一直是暴露点，因为一个节点一旦成为匹配点，则它以后一直保持为匹配点. 所以，i 应当一直标号为外点，每次算法修改对偶变量都包括对 α_i 的修改（减少）. 由于开始时 $\boldsymbol{\alpha}$ 的所有分量是相等的 $\left(\frac{1}{2} \max\limits_{(j,k)\in A} c_{jk}\right)$，所以 α_i 是 $\boldsymbol{\alpha}$ 的所有分量中最小的一个. 因此，$\delta_1 = \alpha_i > 0$，并且显然有 $\delta_1 = \alpha_i \leqslant \frac{1}{2} \max\limits_{(j,k)\in A} c_{jk}$.

(2) 现在证明 $\delta_2 > 0$. 也就是说，需要证明 $\{S_k \in OS^- \mid \beta_k = 0\} = \varnothing$. 由于每次增广（或修改对偶变量）之后，都将展开所有满足 β_k 的花朵（节点奇集）S_k，因此如果当前存在某个满足 $\beta_i = 0$ 的花朵 S_i，则 S_i 一定是在最近的一次增广（或修改对偶变量）之后被发现并被收缩的. 但根据算法，这样的人工节点应当得到外点标号，即 $S_i \in OS^+$. 所以 $\{S_k \in OS^- \mid \beta_k = 0\} = \varnothing$.

(3) 现在证明 $\delta_3 > 0$. 假设此时存在 $(i, j) \in A^{++}$ 满足 $c_{ij}^{\pi} = 0$. 由于所有满足 $c_{ij}^{\pi} = 0$ 的弧都一定属于 N'，所以 (i, j) 属于 N'. 由于 i, j 不属于同一个收缩的花朵，因此一定存在一条增广路，这时不应当进行修改对偶变量的操作. 因此我们证明了 $\{(i, j) \in A^{++} \mid c_{ij}^{\pi} = 0\} = \varnothing$，所以 $\delta_3 > 0$.

(4) 现在证明 $\delta_4>0$. 假设此时存在$(i,j)\in A^+$满足 $c_{ij}^{\pi}=0$. 由于所有满足 $c_{ij}^{\pi}=0$ 的弧都一定属于 N'，所以(i,j)属于 N'. 因此节点 j 可以获得"INNER"标号，标号后$(i,j)\notin A^+$. 因此我们证明了$\{(i,j)\in A^+\mid c_{ij}^{\pi}=0\}=\varnothing$，所以 $\delta_4>0$.

根据 $\delta=\min\{\delta_1,\delta_2,\delta_3,\delta_4\}$，可知 $0<\delta<\frac{1}{2}\max\limits_{(j,k)\in A} c_{jk}$. □

引理 7.3 对于原始可行解 $\boldsymbol{x}$ 和对偶可行解$\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$，对偶变量的修改操作之后 $\boldsymbol{x}$ 和 $\boldsymbol{\pi}$ 仍然是原始可行解和对偶可行解.

证明 由于没有修改 $\boldsymbol{x}$，因此 $\boldsymbol{x}$ 仍然是原始可行的. 用$\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$表示修改之前的对偶变量，$\boldsymbol{\pi}'=(\boldsymbol{\alpha}',\boldsymbol{\beta}')$表示修改之后的对偶变量，我们需要证明$\boldsymbol{\pi}'$是对偶可行的. 根据修改对偶变量的公式和引理 7.2，

$\forall i\in V$，$\alpha_i'\geqslant\alpha_i-\delta\geqslant\alpha_i-\delta_1\geqslant 0$；

$\forall$ 节点奇集 $S_k\in OS$，$\beta_k'\geqslant\beta_k-2\delta\geqslant\alpha_i-2\delta_2\geqslant 0$.

所以，为了证明$\boldsymbol{\pi}'=(\boldsymbol{\alpha}',\boldsymbol{\beta}')$的对偶可行性，只需证明 $\forall(i,j)\in A$，$c_{ij}^{\pi'}\geqslant 0$. 我们分几种情况进行讨论.

(1) $(i,j)\notin A^+\cup A^{++}$，且$(i,j)$不属于已经收缩并标号的节点奇集. 此时根据修改对偶变量的公式，$c_{ij}^{\pi'}\geqslant c_{ij}^{\pi}\geqslant 0$.

(2) $(i,j)\in A^+$. 此时利用 δ 和 δ_4 的定义，$c_{ij}^{\pi'}=c_{ij}^{\pi}-\delta\geqslant c_{ij}^{\pi}-\delta_4\geqslant 0$.

(3) $(i,j)\in A^{++}$. 此时利用 δ 和 δ_3 的定义，$c_{ij}^{\pi'}=c_{ij}^{\pi}-\delta\geqslant c_{ij}^{\pi}-\delta_3\geqslant 0$.

(4) (i,j)属于已经收缩并标号为"OUTER"(外点)的节点奇集. 此时 $i,j\in V^+$，根据修改对偶变量的公式，$c_{ij}^{\pi'}=c_{ij}^{\pi}-\delta-\delta+2\delta=c_{ij}^{\pi}\geqslant 0$.

(5) (i,j)属于已经收缩并标号为"INNER"(内点)的节点奇集. 此时 $i,j\in V^-$，根据修改对偶变量的公式，$c_{ij}^{\pi'}=c_{ij}^{\pi}+\delta+\delta-2\delta=c_{ij}^{\pi}\geqslant 0$.

因此我们证明了$\boldsymbol{\pi}'=(\boldsymbol{\alpha}',\boldsymbol{\beta}')$的对偶可行性. □

引理 7.4 如果原始可行解 $\boldsymbol{x}$ 和对偶可行解$\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$满足式(7.13)和式(7.14)，则对偶变量的修改操作之后 $\boldsymbol{x}$ 和$\boldsymbol{\pi}$ 仍然满足式(7.13)和式(7.14).

证明 用$\boldsymbol{\pi}=(\boldsymbol{\alpha},\boldsymbol{\beta})$表示修改之前的对偶变量，$\boldsymbol{\pi}'=(\boldsymbol{\alpha}',\boldsymbol{\beta}')$表示修改之后的对偶变量.

(1) 要证明式(7.13)成立，只需证明对于任意匹配弧(i,j)，$c_{ij}^{\pi'}=c_{ij}^{\pi}(=0)$. 如果$(i,j)$没有被收缩，则它的两个端点要么都没有标号，要么一个为内点而另一个为外点. 根据修改对偶变量的公式，此时总有 $c_{ij}^{\pi'}=c_{ij}^{\pi}$.

如果(i,j)属于已经收缩并标号的节点奇集，则由前面证明的(4)和(5)知 $c_{ij}^{\pi'}=c_{ij}^{\pi}$.

如果(i,j)属于已经收缩但没有标号的节点奇集，则根据修改对偶变量的公式知 $c_{ij}^{\pi'}=c_{ij}^{\pi}$.

(2) 要证明式(7.14)成立，只需证明对于任意节点奇集，如果 S_k 不是花朵(即 S_k 中匹配弧的条数$<s_k$)，则 $\beta_k'=\beta_k(=0)$. 这是显然的，因为 S_k 不是花朵时自然不会对它进行收缩，根据修改对偶变量的公式，此时 $\beta_k'=\beta_k$. □

引理 7.5 对偶变量的修改操作之后，交错森林中的边(i, j)仍然满足$c_{ij}^{\pi}=0$.

证明 用$\boldsymbol{\pi}=(\boldsymbol{\alpha}, \boldsymbol{\beta})$表示修改之前的对偶变量，$\boldsymbol{\pi}'=(\boldsymbol{\alpha}', \boldsymbol{\beta}')$表示修改之后的对偶变量，我们需要证明$c_{ij}^{\pi'}=c_{ij}^{\pi}=0$. 由于$(i, j)$要么位于某个被标号的花朵内，要么它的两个端点中一个为内点而另一个为外点，根据引理7.4的讨论(1)，此时总有$c_{ij}^{\pi'}=c_{ij}^{\pi}$. □

引理 7.6 在上述算法中，每两次相邻增广操作之间最多执行$5n/3$次修改对偶变量的操作.

证明 我们对δ的不同取值分别讨论：

(1) $\delta=\delta_1$. 由于所有暴露点i一定属于V^+且其对偶变量$\alpha_i=\delta_1$，所以$\delta=\delta_1$时执行当前修改对偶变量的操作以后将得到最优解，算法结束.

(2) $\delta=\delta_2$. 此时，修改对偶变量一定使得某个被收缩并标号为内点的奇集S_k的对偶变量从非零修改为0(即$\beta_k'=0<\beta_k$). 在一次增广操作之后，被收缩并标号为内点的奇集最多为$n/3$个；而在两次相邻增广操作之间，被收缩的奇集一定获得外点标号. 所以$\delta=\delta_2$的情况在两次相邻增广操作之间最多出现$n/3$次.

(3) $\delta=\delta_3$. 此时，修改对偶变量一定使得某条两个端点均被标号为外点的弧(i, j)所对应的c_{ij}^{π}从非零修改为0(即$c_{ij}^{\pi'}=0<c_{ij}^{\pi}$). 也就是说$i$，$j$都是外点，并且$(i, j)$加入$N'$中. 此时，不是发现一条增广路而进行增广操作，就是发现一个花朵而实施收缩操作. 发现一个花朵而实施收缩操作时，该花朵对应的人工节点一定获得外点标号，所以该花朵对应的对偶变量β_k在下一次增广操作之前只能增加，所以不会有机会被展开. 因此$\delta=\delta_3$的情况在两次相邻增广操作之间最多出现$n/3$次.

(4) $\delta=\delta_4$. 此时，修改对偶变量一定使得某条弧(i, j)所对应的c_{ij}^{π}从非零修改为0(即$c_{ij}^{\pi'}=0<c_{ij}^{\pi}$)，且$(i, j)$中一个节点(不妨设为$i$)是外点，另一个节点$(j)$没有标号. (i, j)加入N'中以后，j获得内点标号. 由于最多有n个节点没有标号，因此$\delta=\delta_3$的情况在两次相邻增广操作之间最多出现n次.

综上所述，每两次相邻增广操作之间最多执行$5n/3$次修改对偶变量的操作. □

根据以上引理，又考虑到算法中总增广操作次数不超过n次，每完成一次修改对偶变量的操作(包括计算δ等)的复杂度为$O(m)$，所以非二部赋权匹配问题的原始-对偶算法时间复杂度为$O(mn^2)$. 可以将前面这些结论归纳成下面的定理.

定理 7.3 非二部赋权匹配问题的原始-对偶算法是正确的，并且其时间复杂度为$O(mn^2)$. □

例 7.4 在图7.5(a)所示的网络中，弧$(e_1, e_2, \cdots, e_9)$上的权向量为$\boldsymbol{c}=(8, 9, 8, 7, 9, 4, 5, 2, 1)$. 计算最大赋权匹配.

计算过程如下：

(1) 首先初始化：

$$x_{ij}=0, \quad \forall (i,j)\in A,$$

$$\alpha_i=\frac{1}{2}\max_{(j,k)\in A} c_{jk}=4.5, \quad \forall i\in V.$$

$$\beta_k=0, \quad \forall \text{节点奇集 } S_k\in OS,$$

$$\boldsymbol{c}_{ij}^{\pi}=(1,0,1,2,0,5,4,7,8).$$

(2) 此时 N'如图 7.5(b)所示.

在 N'中找到增广路 2-3，增广后得到 $x_{23}=1$. 继续在 N'中找到增广路 4-5，增广后得到 $x_{45}=1$. 此时 $M=\{e_2, e_5\}$. 分别对 N' 中的暴露点 1，6，7 进行检查，得 N' 中各节点的标号如图 7.5(c)所示(括号中第 1 部分表示标号，第 2 部分表示 pred,即从何处获得标号).

(3) 由于此时找不到增广路，因此确定 $V^+=\{1,6,7\}$，$V^-=\{2,3,4,5\}$，其他集合都是$\varnothing$. 此时计算

$$\delta_1=\min\{\alpha_1,\alpha_6,\alpha_7\}=4.5,$$

$$\delta_2=\infty,$$

$$\delta_3=\frac{1}{2}c_{67}^{\pi}=4,$$

$$\delta_4=\min\{c_{12}^{\pi},c_{46}^{\pi},c_{47}^{\pi},c_{57}^{\pi}\}=1,$$

$$\delta=\min\{\delta_1,\delta_2,\delta_3,\delta_4\}=\delta_4=1,$$

$$\boldsymbol{\alpha}=(3.5,4.5,4.5,4.5,4.5,3.5,3.5),$$

$$\beta_k=0 \text{ 不变}(\forall \text{ 节点奇集 } S_k\in OS),$$

$$\boldsymbol{c}_{ij}^{\pi}=(0,0,1,2,0,4,3,6,6).$$

此时应当将 $e_1=(1,2)$加入 N' 中，继续标号得各节点的标号如图 7.5(d)所示.

(4) 由于此时仍然找不到增广路，因此确定 $V^+=\{1,3,6,7\}$，$V^-=\{2\}$，$V^-=\{4,5\}$，其他集合都是$\varnothing$. 此时计算

$$\delta_1=\min\{\alpha_1,\alpha_3,\alpha_6,\alpha_7\}=3.5,$$

$$\delta_2=\infty,$$

$$\delta_3=\frac{1}{2}c_{67}^{\pi}=3,$$

$$\delta_4=\min\{c_{34}^{\pi},c_{35}^{\pi},c_{46}^{\pi},c_{47}^{\pi},c_{57}^{\pi}\}=1,$$

$$\delta=\min\{\delta_1,\delta_2,\delta_3,\delta_4\}=\delta_4=1,$$

$$\boldsymbol{\alpha}=(2.5,5.5,3.5,4.5,4.5,2.5,2.5),$$

$$\beta_k=0 \text{ 不变}(\forall \text{ 节点奇集 } S_k\in OS),$$

$$\boldsymbol{c}_{ij}^{\pi}=(0,0,0,1,0,3,2,5,4).$$

此时应当将 $e_3=(3,4)$加入 N'中，继续标号得各节点的标号如图 7.5(e)所示.

(5) 由于此时仍然找不到增广路，因此确定 $V^+=\{1,3,5,6,7\}$，$V^-=\{2,4\}$，其他集合都是$\varnothing$. 此时计算

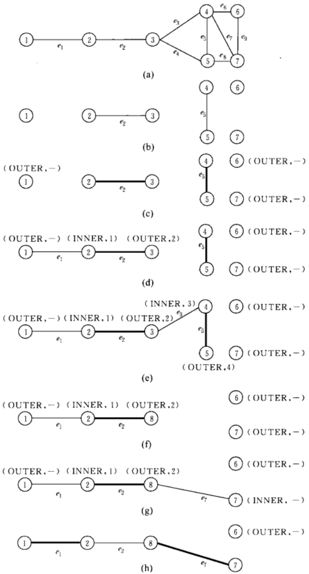


图 7.5 非二部图赋权匹配的例子(例 7.4)

$$\delta_1 = \min\{\alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_7\} = 2.5,$$
$$\delta_2 = \infty,$$
$$\delta_3 = \frac{1}{2}\min\{c_{35}^{\pi}, c_{57}^{\pi}, c_{67}^{\pi}\} = 0.5\min\{1, 5, 4\} = 0.5,$$
$$\delta_4 = \infty,$$
$$\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4\} = \delta_3 = 0.5,$$
$$\boldsymbol{\alpha} = (2, 6, 3, 5, 4, 2, 2),$$
$$\beta_k = 0 \text{ 不变}(\forall \text{ 节点奇集 } S_k \in OS),$$
$$\boldsymbol{c}_{ij}^{\pi} = (0, 0, 0, 0, 0, 3, 2, 4, 3).$$

此时应当将 $e_4=(3, 5)$加入 N'中，但继续标号时发现节点 3 和节点 5 的标号都是“OUTER”，因此得到了一个花朵 $W_1=\{3, 4, 5\}$.

(6) 收缩花朵 $S_1=\{3, 4, 5\}$，记该节点标号为 8. 由此得各节点的标号如图 7.5(f)所示.

(7) 由于此时仍然找不到增广路，因此确定 $V^+=\{1, 3, 4, 5, 6, 7\}$，$V^-=\{2\}$，其他集合都是$\varnothing$. 此时计算

$$\delta_1 = \min\{\alpha_1, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7\} = 2,$$
$$\delta_2 = \infty,$$
$$\delta_3 = \frac{1}{2}\min\{c_{46}^{\pi}, c_{47}^{\pi}, c_{57}^{\pi}, c_{67}^{\pi}\} = 0.5\min\{3, 2, 4, 3\} = 1,$$
$$\delta_4 = \infty,$$
$$\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4\} = \delta_3 = 1,$$
$$\boldsymbol{\alpha} = (1, 7, 2, 4, 3, 1, 1),$$
$$\beta_1 = 2, \beta_k = 0 \text{ 不变}(\forall \text{ 节点奇集 } S_k \in OS, k \neq 1),$$
$$\boldsymbol{c}_{ij}^{\pi} = (0, 0, 0, 0, 0, 1, 0, 2, 1).$$

此时应当将 $e_7=(4, 7)=(8, 7)$加入 N' 中，继续标号，如图 7.5(g)所示.

(8) 在 N'中找到增广路 1-2-8-7，增广后得到 $x_{12}=1$，$x_{23}=0$，$x_{35}=1$，$x_{45}=0$，$x_{47}=1$. 此时 $M=\{e_1, e_4, e_7\}$，去掉所有节点的标号，并重新开始标号，得到 N'中各节点的标号如图 7.5(h)所示.

(9) 由于此时找不到增广路，因此确定 $V^+=\{6\}$，其他集合都是$\varnothing$. 此时计算

$$\delta_1 = \alpha_6 = 1,$$
$$\delta_2 = \infty,$$
$$\delta_3 = \infty,$$
$$\delta_4 = \min\{c_{46}^{\pi}, c_{67}^{\pi}\} = \min\{1, 1\} = 1,$$
$$\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4\} = \delta_1 = \delta_4 = 1,$$
$$\boldsymbol{\alpha} = (1, 7, 2, 4, 3, 0, 1),$$
$$\beta_1 = 2, \beta_k = 0 \text{ 不变}(\forall \text{ 节点奇集 } S_k \in OS, k \neq 1).$$

此时满足条件(7.18)，因此得到了最优解 $M=\{e_1, e_4, e_7\}$. □

复杂度分析：前面说过，$|OS|=2^{n-1}-n$，所以$\boldsymbol{\alpha}$ 的维数是 n 的指数函数. 幸运的是，我们在算法中并不需要同时对$\boldsymbol{\alpha}$ 的所有分量进行操作，因此最后得到的算法的复杂度为 $O(mn^2)$，仍然是多项式时间算法.

从历史上看，非二部赋权匹配算法也是由 Edmonds 于 1965 年首先提出的，当时给出的算法的复杂度为 $O(n^4)$. 此后，许多研究将算法的复杂度不断改进为 $O(mn^2)$，$O(n^3)$等. 目前求解非二部赋权匹配问题的复杂度最低的强多项式算法的复杂度最好为 $O(mn+n^2\log n)$. 对于完全图，由于 $m=O(n^2)$，所以复杂度为 $O(n^3)$. 而且可以看出，这一结果与二部赋权匹配算法的复杂度是一样的. 从这个角度来看，二部赋权匹配与非二部赋权匹配在算法的复杂度上没有本质的区别.

练 习 题

1. 某家汽车公司上午和下午各有 n 次班车运行，分别由 n 个司机驾驶(每个司机上下午各出车一次). 班车的运行时间有长有短，因此当某个司机上下午出车时间之和超过 8 h 以后，公司必须付给该司机一定的加班费(元/h). 如果你是公司经理，你如何安排司机出车，使得公司付出的总的加班费最少？这一问题是否可以用匹配问题的模型来描述？

2. 在战争中，经常需要确定空中多个运动物体(如飞机和导弹等)的运动速度和方向. 一种可能的方法是在较短的时间间隔内两次拍摄空中运动物体的照片，然后确定两张照片中物体的对应关系，以便计算物体的运动速度和方向. 为了确定两张照片中物体的对应关系，一般可以依据下面的假设：两张照片中对应物体之间的距离的平方和最小. 请用匹配问题的模型对这一问题进行建模.

3. 在装箱问题中，如果已知待装物体的尺寸都大于 1/3 (假设箱子的尺寸为 1)，证明此时装箱问题可以转化为匹配问题.

4. 证明：在一个 8×8 的正方形中删去两个位于对角上的1×1小正方形后，不能用 1×2 的长方形正好覆盖.

5. 考虑如下问题：假设有 n 个男人和 n 个女人，每人都希望从 n 个异性中选择一位自己的配偶. 假设每人都对 n 个异性根据自己的偏好进行了排序，以此作为选择配偶的基础. 当给定一种婚配方案(即给每人指定一个配偶)后，如果存在一个男人和一个女人不是互为配偶，但该男人喜欢该女人胜过其配偶，且该女人喜欢该男人也胜过其配偶，则该婚配方案称为不稳定的. 安排稳定的婚配方案的问题称为稳定婚配问题，试设计有效算法求解该问题，并分析你给出的算法的计算复杂度.

6. 利用练习题 5 的结果，求解如下 5 男 5 女的稳定婚配问题：用矩阵 $\boldsymbol{M}$ 表示男人对女人的排序(序号越大表示喜欢程度越高)，矩阵 $\boldsymbol{W}$ 表示女人对男人的排序，给定 $\boldsymbol{M}$ 和 $\boldsymbol{W}$ 如下：

$$M=\begin{bmatrix}3&5&2&1&4\\4&3&5&1&2\\4&1&3&2&5\\1&3&2&5&4\\4&2&3&1&5\end{bmatrix},\quad W=\begin{bmatrix}5&4&3&1&2\\5&1&3&2&4\\5&4&1&3&2\\5&3&1&2&4\\5&3&2&1&4\end{bmatrix}.$$

7. 在如图 7.6 所示网络图中，计算最大匹配.

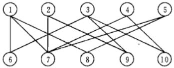


图 7.6 第 7 题图

8. 在如图 7.7 所示网络图中，计算最大匹配.

9. 在如图 7.8 所示网络图中，计算最大匹配.

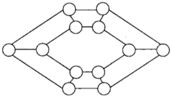


图 7.7 第 8 题图

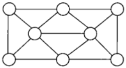


图 7.8 第 9 题图

10. 在一个 $n\times n$ 的方阵中，两两不同行不同列的 n 个元素的集合称为该方阵的一条对角线. 在下面的方阵中求一条对角线，使得其对角线上元素的和最大.

$$\begin{matrix}4&5&3&1&0\\2&2&2&4&3\\0&4&1&5&0\\0&6&4&5&5\\4&2&3&4&2\end{matrix}$$

11. 已知不同工人(行)完成不同工作(列)的利润矩阵如下，求解指派问题，使得总的利润最大.

$$\begin{matrix}1&3&2&6&0\\4&2&3&8&3\\8&4&1&5&0\\3&5&4&8&8\\2&6&9&5&2\end{matrix}$$

12. 计算图 7.9 中的最大赋权匹配.

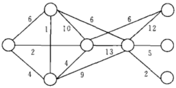


图 7.9　第 12 题图

13. 对于无向图 $N=(V, A)$中的匹配 M，如果 $\forall (i, j)\in A\backslash M$，$M\cup\{(i, j)\}$不是一个匹配，则称 M 为图 N 的极大匹配. 试设计一个复杂度为 $O(m)$的算法求解极大匹配问题，并证明极大匹配的基数至少是最大匹配的基数的一半.

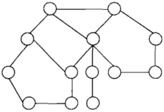
14. 一个无向图中的弧覆盖是指图中的一个弧子集，该弧子集中的所有弧的端点集合等于无向图的节点集合. 无向图中的弧数最少的弧覆盖称为最小弧覆盖. 试设计有效算法求解最小弧覆盖问题，并分析你给出的算法的计算复杂度.

15. 在如图 7.10 所示的街道网络中，站在一条街道上的警察可以监视这条街道两头的路口. 请你安排尽可能少的警察，使得所有路口都可以得到监视.

16. 无向网络的所有最大基数匹配中，权最大的一个称为网络的最大权最大匹配，权最小的一个称为网络的最小权最大匹配. 试利用最大赋权匹配算法，设计有效算法求解最大权最大匹配和最小权最大匹配问题，并分析你给出的算法的计算复杂度.

17. 请设计一个求解无向网络上的中国邮递员问题的有效算法(中国邮递员问题参见例 1.5)，并分析你给出的算法的计算复杂度.

18. 在如图 7.11 所示的街道网络中，弧上的权表示街道的长度. 请计算一条最优邮路(即为邮递员设计一条最佳投递路线)：


图 7.10　第 15 题图

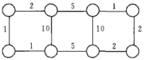


图 7.11　第 18 题图

索引及英文关键词

（1）按笔划顺序排列.
（2）$m.n$ 指第 m 章第 n 节 .

八划

九划

十划

十一划

十二划

参考文献

[1] Ahuja R K, Magnanti T L and Orlin J B. Network Flows: Theory, Algorithms, and Applications. Englewood Cliffs, New Jersey: Prentice Hall, 1993

[2] Balakrishnan V K, Network Optimization. London: Chapman & Hall, 1995

[3] Bazaraa M S, Jarvis J J and Sherali H D. Linear Programming and Network Flows. 2nd ed. New York: John Wiley & Sons, 1990

[4] Bertsekas D P. Linear Network Optimization: Algorithms and Codes. Massachusetts, Cambridge: MIT Press, 1991

[5] Evans J R, Minieka E. Optimization Algorithms for Networks and Graphs. 2nd ed. New York: Marcel Dekker Inc., 1992

[6] Garey M R, Johnson D S. Computers and Intractability : a Guide to the Theory of NP-Completeness. San Francisco : W. H. Freeman and Co., 1979（中译本：《计算机和难解性：NP-C 理论导论》. 张立昂等译. 北京：科学出版社，1987）

[7] Kleinberg J and Tardos É. Algorithm Design. Pearson Education Inc., 2005

[8] Lawler E L. Combinatorial Optimization : Networks and Matroids. Holt. New York: Rinehart and Winston, 1976

[9] Nemhauser G L, Wolsey L A. Integer and Combinatorial Optimization. New York: John Wiley & Sons, 1988

[10] Papadimitriou C H, Steiglitzk. Combinatorial Optimization: Algorithms and Complexity. Englewood Cliffs, New Jersey: Prentice Hall, 1982（中译本：《组合最优化：算法与复杂性》，刘振宏，蔡茂诚译. 北京：清华大学出版社，1988）

[11] Tarjan R E. Data Structures and Network Algorithms. Philadelphia, Pennsylvania: SIAM, 1983

[12] 刘家壮，徐源. 网络最优化. 北京：高等教育出版社，1991

[13] 卢开澄. 计算机算法导引 ：设计与分析. 北京：清华大学出版社，1996

[14] 马仲蕃. 线性整数规划的数学基础. 北京：科学出版社，1995

[15] 钱颂迪主编. 运筹学. 北京：清华大学出版社，1990

[16] 田丰，马仲蕃. 图与网络流理论. 北京：科学出版社，1987

[17] 谢政，李建平. 网络算法与复杂性理论. 长沙：国防科技大学出版社，1995

[18] 邢文训，谢金星. 现代优化计算方法. 第 2 版. 北京：清华大学出版社，2005